Results of SEI Independent Research and Development Projects and Report on Emerging Technologies and Technology Trends

Sagar Chaki
Rosann W. Collins
Peter Feiler
John Goodenough
Aaron Greenhouse
Jorgen Hansson
Alan R. Hevner
John Hudak
Angel Jordan
Rick Kazman
Richard C. Linger
Mark G. Pleszkoch
Stacy J. Prowell
Natasha Sharygina
Kurt C. Wallnau
Gwen Walton
Chuck Weinstock
Lutz Wrage

December 2005

TECHNICAL REPORT
CMU/SEI-2005-TR-020
ESC-TR-2005-020

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Results of SEI Independent Research and Development Projects and Report on Emerging Technologies and Technology Trends

CMU/SEI-2005-TR-020
ESC-TR-2005-020

Sagar Chaki
Rosann W. Collins
Peter Feiler
John Goodenough
Aaron Greenhouse
Jorgen Hansson
Alan R. Hevner
John Hudak
Angel Jordan
Rick Kazman
Richard C. Linger
Mark G. Pleszkoch
Stacy J. Prowell
Natasha Sharygina
Kurt Wallnau
Gwen Walton
Chuck Weinstock
Lutz Wrage

December 2005

SEI Director's Office

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Abstract ..... **ix**

Introduction **1**
1.1 Purpose of the SEI Independent Research and Development Program 1
1.2 Overview of IR&D Projects 2
1.3 Purpose of Technology Scouting 2

2 Architecture-Based Self-Adapting Systems **3**
2.1 Purpose 3
2.2 Background 4
2.3 Approach 5
2.4 Collaborations 6
2.5 Evaluation Criteria 6
2.6 Results 7
2.7 Publications and Presentations 7
2.7.1 Publications 7
2.7.2 External presentations: 8
2.7.3 References 8

3 The Impact of Architecture Concurrency on Performance Engineering **11**
3.1 Purpose 11
3.2 Background 12
3.3 Approach 13
3.4 Collaborators 13
3.5 Evaluation Criteria 14
3.6 Results 14
3.7 Course and Publications 16
3.8 References 16

4 The Impact of Function Extraction Technology on Next-Generation Software Engineering **19**
4.1 Purpose 19
4.2 Background 20

4.2.1 The Idea of Function Extraction 20
4.2.2 Fundamentals of Program Behavior Calculation 22
4.3 Approach 23
4.4 Collaborations 25
4.5 Evaluation Criteria 25
4.6 Results 25
4.7 References 27

5 Issues in Scalability 31
5.1 Purpose 31
5.2 Background 32
5.3 Approach 32
5.4 Collaborations 33
5.5 Evaluation Criteria 33
5.6 Results 33
5.6.1 What is Scalability? 34
5.6.2 Achieving Scalability Involves Making Tradeoffs 35
5.7 Conclusion 35
5.8 References 36

6 Proof-Carrying Code 37
6.1 Purpose 37
6.2 Background 38
6.3 Approach 38
6.4 Collaborations 38
6.5 Evaluation Criteria 39
6.6 Results 39
6.6.1 Survey of the State of the Research 39
6.6.2 Prototype Certifying Model Checker 42
6.6.3 Separation Logic for Predicate Abstraction 43
6.7 References 44

7 Verification of Evolving Software via Component Substitutability Analysis 47
7.1 Introduction 47
7.2 Model Checking 49
7.3 The Process of Model Checking 50
7.4 Current Research in Software Model Checking 50
7.4.1 Compositional Reasoning 51
7.4.2 Abstraction 52

7.4.3 Counterexample-Guided Abstraction Refinement (CEGAR) 52
7.5 Verification of Evolving Software 54
7.6 Implementation and Experimental Evaluation 55
7.7 Related Work 56
7.8 Conclusion 57
7.9 References 57

8 Emerging Technologies and Technology Trends 63
8.1 Introduction 63
8.1.1 Reducing Software Defects to Improve Security 64
8.1.2 Organization of this Report 65
8.2 Technology Scouting of Work at Carnegie Mellon University and Other Institutions Worldwide Relevant to SEI 66
8.2.1 Advances in Software Architecture 67
8.2.2 Aspect-Oriented Programming (AOP) and Aspect-Oriented Software Development (AOSD) 72
8.2.3 Autonomic Application Software 74
8.2.4 Verification of Autonomous Systems 75
8.2.5 Proof-Carrying Code 77
8.2.6 The ConCert Project 77
8.2.7 The Fox Project 78
8.2.8 Building Certifiably Dependable Software Systems 78
8.3 Technology Scouting in Systems and Software Engineering 79
8.3.1 Introduction 79
8.3.2 2005 Software Process Workshop 86
8.3.3 Agile Software Development 96
8.3.4 International Conferences in Software Engineering 99
8.3.5 Recipients and Title of Most Influential Paper 99

List of Figures

Figure 2-1: The DiscoTect System 5

Figure 4-1: The Basic Concept of Function Extraction 22

Figure 6-1: Archetypal Proof-Carrying Code 40

Figure 7-1: A Small Program with Two Threads of Control 48

Figure 7-2: The CEGAR Framework 53

Figure 7-3: Comparison of Times Required for Original Verification (T_{orig}) and Verification on Upgrade (T_{ug}) by DynamicCheck 55

List of Tables

Table 4-1: Creation and Loss of Semantic Information in Software Development .. 21

Table 4-2: FX Impacts—Where to Next? 24

Table 6-1: Comparison of Proof Certificate Size SAT vs. Conventional Theorem Provers 43

Abstract

Each year, the Software Engineering Institute (SEI) undertakes several Independent Research and Development (IR&D) projects. These projects serve to (1) support feasibility studies investigating whether further work by the SEI would be of potential benefit, and (2) support further exploratory work to determine whether there is sufficient value in eventually funding the feasibility study work as an SEI initiative. Projects are chosen based on their potential to mature and/or transition software engineering practices, develop information that will help in deciding whether further work is worth funding, and set new directions for SEI work. This report describes the IR&D projects that were conducted during fiscal year 2005 (October 2004 through September 2005). In addition, this report provides information on what the SEI has learned in its role as a technology scout for developments over the past year in the field of software engineering.

1 Introduction

This document briefly describes the results of the independent research and development projects conducted at the Carnegie Mellon Software Engineering Institute (SEI) during the 2004–05 fiscal year. It also provides information about what the SEI has learned in its role as a technology scout for developments over the past year in the field of software engineering.

1.1 Purpose of the SEI Independent Research and Development Program

SEI independent research and development (IR&D) funds are used in two ways: (1) to support feasibility studies investigating whether further work by the SEI would be of potential benefit and (2) to support further exploratory work to determine if there is sufficient value in eventually funding the feasibility study work as an SEI initiative. It is anticipated that each year there will be three or four feasibility studies and that one or two of these studies will be further funded to lay the foundation for the work possibly becoming an initiative.

Feasibility studies are evaluated against the following criteria:

- Mission criticality: To what extent is there a potentially dramatic increase in maturing and/or transitioning software engineering practices if work on the proposed topic yields positive results? What will the impact be on the Department of Defense (DoD)?
- Sufficiency of study results: To what extent will information developed by the study help in deciding whether further work is worth funding?
- New directions: To what extent does the work set new directions as contrasted with building on current work? Ideally, the SEI seeks a mix of studies that build on current work and studies that set new directions.

At a DoD meeting in November 2001, the SEI's DoD sponsor approved a set of thrust areas and challenge problems to provide long-range guidance for the SEI research and development program, including its IR&D program. The thrust areas are survivability/security, interoperability, sustainability, software R&D, metrics for acquisition, acquisition management, and commercial off-the-shelf products. The IR&D projects conducted in FY2005 were based on these thrust areas and challenge problems.

1.2 Overview of IR&D Projects

The following research projects were undertaken in FY2005:

- Architecture-based Self Adapting Systems
- Assessing and Demonstrating the Readiness of Proof Carrying Code for Obtaining Objective Trust in Software Components
- A Feasibility Study of Automated Program Behavior Computation for Next-Generation Software Engineering
- Impact of Architecture Concurrency on Performance Engineering
- Software Scalability
- Verification of Evolving Software via Component Substitutability Analysis

These projects are described in detail in this technical report.

1.3 Purpose of Technology Scouting

Technology scouting has always been an implicit activity of the Software Engineering Institute and is embedded in the SEI's mission of technology transition. Because of the institute's small size relative to other research institutions, the SEI applies the most leverage to its active initiatives, but it also watches for other emerging technologies, in the U.S. and internationally. The SEI has been asked to report on the state of the art of software technologies—those that are pushing the frontiers of the SEI's current programs and initiatives and also those that transcend them.

2 Architecture-Based Self-Adapting Systems

Rick Kazman

2.1 Purpose

A well-defined software architecture is critical for the success of complex software systems. Such a definition provides a high-level view of a system in terms of its principal runtime components (e.g., clients, servers, databases), their interactions (e.g., remote procedure call, event multicast, piped streams), and their properties (e.g., throughputs, latencies, reliabilities). As an abstract representation of a system, an architecture permits many forms of high-level inspection and analysis, allowing the architect to determine if a system's design will satisfy its critical quality attributes. Consequently, over the past decade, considerable research and development has gone into the development of notations, tools, and methods to support architectural design. However, despite considerable progress in developing an engineering basis for software architecture, a persisting difficult problem is determining whether a system as implemented has the architecture as designed. Without some form of consistency guarantees, the validity of any architectural analysis will be suspect, at best, and completely erroneous, at worst.

In addition, an increasingly important requirement for software-based systems is the ability to adapt themselves at runtime to handle such things as changing resources, changing user needs and demands, and system faults. In the past, systems that supported such self-repair were rare, confined mostly to domains like telecommunications switches or deep space control software, where taking a system down for upgrades was not an option, and where human intervention was not always feasible. However, today more and more systems have this requirement, including e-commerce systems and mobile embedded systems.

For systems to benefit from having a well-defined software architecture, there must be a way of ensuring that the implementation conforms to its architecture. And for systems to adapt themselves at runtime, one of the essential ingredients is self-reflection: a system must know its architecture, its current level of various quality attributes (such as performance, security, availability, and usability), and it must be able to identify opportunities for improving its own quality attribute behavior by changing its properties or even changing its structure. In this work we have built upon the successes of our initial SEI-funded exploratory study (conducted during fiscal year 2004) by showing how we can use software architecture descriptions discovered at runtime as a basis for system validation, system reflection, and self-

adaptation. This work extends the DiscoTect framework [Yan 04a, 04b] to extract quality attribute information from running systems, and to reason about this information.

Traditionally software systems have operated in relatively stable, fixed environments (such as a desktop), and could be taken down for maintenance, upgrading, or replacement. However, increasingly software systems must function in environments where resources (such as bandwidth or power) change rapidly, where their resource demands are difficult to predict in advance, and where they must interact with potentially faulty components, services, and threats not under their control. Despite this, such systems must operate continuously and provide the highest quality of service possible. Thus such systems must take responsibility for their own health and welfare, adapting at runtime to handle threats, errors, changing resources, and varying user needs.

Software engineers currently have few tools or techniques at their disposal to create such self-adaptive systems reliably, flexibly, and at low cost. Most existing techniques rely on low-level mechanisms, such as exceptions and timeouts. But these generally provide little help in allowing a system to determine the true source of problems, or in deciding what to do about them. Moreover, they are ineffective at dealing with softer problems, such as gradual performance degradation, or for recognizing opportunities to improve behavior even when things are not broken.

2.2 Background

Our work is mostly related to other approaches for dynamic analysis of a system. A number of techniques and tools have been developed to extract information from a running system. These include instrumenting the source code to produce trace information and manipulating runtime artifacts to get the information (e.g., as described by Balzer and Goldman [Balzer 99] and Wells and Pazandak [Wells 01]). There are many technologies available for monitoring systems, and we build on those. However, they do not by themselves solve the hard problem of mapping from code to more abstract models. In previous work, we developed an infrastructure for doing certain kinds of abstraction [Garlan03]. However, this approach was limited to observing properties of a system and reflecting them in a pre-constructed architectural model. In this work, we show how to create that model.

The work by Dias and Richardson [Dias 03] uses an extensible markup language (XML)-based language to describe runtime events and use patterns to map these events into high-level events. Analyzing these events to determine architectural structure is not addressed. In addition, a simple static mapping from low-level system events to high-level events has limited expressiveness. For example, it cannot handle the case where the event analyzer initially has an interest in one set of events, but then changes its interest after the initial events have occurred. Also it doesn't provide a way of specifying event correlations or mapping a series of correlated low-level events to a single high-level event—a crucial capability needed when discovering the architecture of a system. Kaiser and colleagues use a collection of temporal

state machines to perform pattern matching against runtime events [Kaiser 03]. Our approach is similar, but it makes architectural styles or patterns explicit.

A number of researchers have investigated the problem of presenting dynamic information to an observer. For example, some researchers present information about variables, threads, activations, object interactions, and so forth [Reiss 03, Walker 00, and Zeller 01]. Ernst and colleagues show how to dynamically detect program invariants by examining values computed during a program execution and by looking for patterns and relationships among them [Ernst 01]. This is somewhat different from detecting architectural structure.

Madhav [Madhav 96] describes a system that allows Ada 95 programs to be monitored dynamically to check conformance to a Rapide architectural specification [Luckham96]. His approach requires the source code to be annotated so that it can be transformed to produce events to construct the architecture. In contrast, our approach does not require access to the source code, and it does not rely on explicit architectural construction directives to be embedded in the code as does the approach used by Aldrich and colleagues [Aldrich 02].

2.3 Approach

We are furthering our exploration into a new paradigm for software systems that promises to solve this problem. The underlying idea is to associate with each software system a reflective model that allows a system to reason about its own quality attribute behavior at runtime, and take action to modify its own structure and behavior when necessary [Garlan 03]. Specifically, we are using architectural models for this purpose. By reflecting the current state of a system as an architectural model that exposes only the main components, interactions, and their high-level properties, a system can much more easily understand what its current state is and take necessary actions.

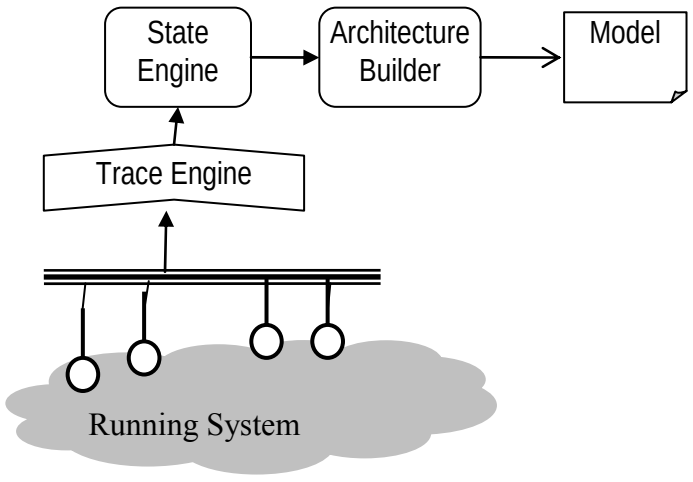

Figure 2-1: The DiscoTect System

A critical step toward achieving this vision is the ability to know exactly what the architecture of a running system is. In our initial study we made great strides toward solving this problem, creating the DiscoTect Architecture, as shown in Figure 2-1. Using DiscoTect we are able to: observe a system's runtime behavior, interpret that runtime behavior in terms of architecturally meaningful events, and represent the resulting architecture. DiscoTect has proven itself capable of extracting the architectures of systems written in Java and C/C++, and includes a wide variety of built-in architectural styles [Yan 04b]. But several obstacles must still be overcome before we can turn this promising foundation into a usable, efficient tool for software engineers. This exploratory study was aimed at maturing DiscoTect, providing a stronger foundation for it, and linking the results of architectural extraction to quality attribute models.

2.4 Collaborations

Collaborators on this project include the following:

SEI:

- Rick Kazman
- William O'Brien

External:

- David Garlan (Carnegie Mellon University/Institute for Software Research International [ISRI] faculty)
- Jonathan Aldrich (Carnegie Mellon/ISRI faculty)
- Bradley Schmerl (Carnegie Mellon/ISRI system scientist)
- Hong Yan (Carnegie Mellon/Computer Science third-year PhD student)
- Owen Chang (Carnegie Mellon/Computer Science fourth-year PhD student)

2.5 Evaluation Criteria

We set forth the following evaluation criteria in our original proposal:

- At least one government and one commercial organization will work with us to have their systems analyzed.
- At least one journal or conference paper will be published on this research.
- At least one technical report will be published on this approach.
- Clear guidance on the feasibility of the approach for future SEI investment and involvement will be written.

2.6 Results

We have made significant strides in maturing DiscoTect in the past year:

- We have formalized the underlying language that expresses the mapping from extracted low-level "tracing" events to architecturally significant events.
- We have created a formal model for the state machine underlying DiscoTect, using Colored Petri Nets.
- We have improved the usability of DiscoTect, so that it can be more easily transferred to practicing software engineers. In particular we have made the DiscoSTEP language more regular, and we have enhanced the layout of the resulting reverse-engineered architecture, using yFiles, a commercial graph-layout package.
- We have made DiscoTect more robust and made it easier to integrate with other software tools. DiscoTect now uses JMS (Java Message Service) to communicate with other tools. DiscoTect gets implementation events from JMS and outputs architecture events to JMS.
- Using this infrastructure we have integrated DiscoTect with other tools, such as AcmeStudio, so that we can interact with the results, visualize the results, and run off-line analyses.
- We have improved the performance of DiscoTect. In particular we have optimized the event-processing algorithms.
- Finally, we have completed three case studies, one of which was on a commercial system:
  - pipe and filter system (student project)
  - adaptive architecture for mobile simulation (research system)
  - JBoss/J2EE (commercial system). Our analysis found a substantial architectural deviation in Sun's implementation of the Duke's Bank system in J2EE.

2.7 Publications and Presentations

2.7.1 Publications

- "Discovering Architectures from Running Systems Uusing Colored Petri Nets." Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. *Transactions on Software Engineering*. Submitted for publication, 2005.
- "DiscoTect: A System for Discovering Architectures from Running Systems (Demonstration)." Bradley Schmerl, Hong Yan, and David Garlan. *The 2005 Joint European Software Engineering Conference and ACM SigSoft Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, September 2005.
- *Discovering Architectures from Running Systems: Lessons Learned*. Hong Yan, Jonathan Aldrich, David Garlan, Rick Kazman, and Bradley Schmerl. Software Engineering Institute Technical Report, CMU-SEI-2004-TR-016, 2004.

- "DiscoTect: A System for Discovering Architectures from Running Systems." Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman, *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, May 2004.

2.7.2 External presentations:

- Research demonstration at *Joint 2005 ACM SIGSOFT Foundations of Software Engineering and European Software Engineering Conferences*, Lisbon, Portugal, September 2005.

2.7.3 References

URLs are valid as of the publication date of this document.

[Aldrich 02] Aldrich, J.; Chambers, C.; & Notkin, D. "ArchJava: Connecting Software Architecture to Implementation." *Proceedings of the 24th International Conference on Software Engineering*. 2002.

[Balzer 99] Balzer, R.M. & Goldman, N.M. "Mediating Connectors." *Proceedings of 19th IEEE International Conference on Distributed Computing Systems Workshop on Electronic Commerce and Web-Based Applications*. Austin, TX, 1999.

[Dias 03] Dias, M. & Richardson, D. "The Role of Event Description on Architecting Dependable Systems (extended version from WADS)." *Lecture Notes in Computer Science: Book on Architecting Dependable Systems* (Springer-Verlag), 2003.

[Ernst 01] Ernst, M.D.; Cockrell, J.; Griswold, W.G.; & Notkin, D. "Dynamically Discovering Likely Program Invariants to Support Program Evolution." *IEEE Transactions on Software Engineering*, 27(2), 2001.

[Garlan 03] Garlan, D.; Cheng, S-W; & Schmerl, B. "Increasing System Dependability Through Architecture-Based Self-Repair." *Architecting Dependable Systems*, R. de Lemos, C. Gacek, A. Romanovsky (eds.). Springer-Verlag, 2003.

[Kaiser 03] Kaiser, G.; Parekh, J.; Gross, P.; & Veletto, G. "Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems." *Proceedings of Fifth International Active Middleware Workshop*. 2003.

[Madhav 96] Madhav, N. "Testing Ada 95 Programs for Conformance to Rapide Architectures." *Proceedings of Reliable Software Technologies—Ada Europe 96*, 1996.

[Reiss 03] Reiss, S. "JIVE: Visualizing Java in Action (Demonstration Description)." *Proceedings of 25th International Conference on Software Engineering*, 2003.

[Walker 00] Walker, R.J.; Murphy, G.C.; Steinbok, J.; & Robillard, M.P. "Efficient Mapping of Software System Traces to Architectural Views." *Proceedings of CASCON 2000,* S.A. MacKay and J.H. Johnson (eds.).

[Wells 01] Wells, D. & Pazandak, P. "Taming Cyber Incognito: Surveying Dynamic/Reconfigurable Software Landscapes." *Proceedings of First Working Conference on Complex and Dynamic Systems Architectures*, 2001.

[Yan 04a] Yan, H.; Garlan, D.; Schmerl, B.; Aldrich, J.; & Kazman, R. "DiscoTect: A System for Discovering Architectures from Running Systems." *Proceedings of the 26th International Conference on Software Engineering (ICSE 26)*, (Edinburgh, Scotland), May 2004, 470-479.

[Yan 04b] Yan, H.; Aldrich, J.; Garlan, D.; Kazman, R.; & Schmerl, B. *Discovering Architectures from Running Systems: Lessons Learned* (CMU/SEI-2004-TR-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004.

[Zeller 01] Zeller, A. "Animating Data Structures in DDD." *Proceedings of SIGCSE/SIGCUE Program Visualization Workshop*, 2000.

3 The Impact of Architecture Concurrency on Performance Engineering

Peter Feiler, Aaron Greenhouse, Jorgen Hansson, John Hudak, Lutz Wrage

3.1 Purpose

Concurrency is key to the performance of many embedded and mission-critical systems. Control systems must process sensor data from many sources and drive the control of a physical system through many actuators. Mission-critical systems must process information from many data sources, maintain a global situational image, and provide individual responses in a timely manner. When such systems require improved performance—in areas such as throughput and response time, for example—an increase in computational resources does not always result in the desired performance gain. The reason is that system architectures and their components have an inherent degree of concurrency that places constraints on performance gains. When analyzing the performance of such systems it is therefore essential to understand their architectural concurrency limitations, and to understand how to overcome them if the performance requirements are not met. This requires performance modeling tools to predict performance and capabilities to generate such performance models from architecture descriptions, as well as an understanding of concurrency constraints imposed by specific system architectures and how to overcome them.

The purpose of this project is to investigate the feasibility of characterizing the degree of concurrency in different system architectures and to predict the impact of such concurrency constraints on system performance, in particular on system latency and throughput. An understanding of this impact of architecture concurrency on performance provides a basis for predictable improvement of system performance through systematic engineering changes to the system architecture. The intent of this project is to demonstrate the impact of concurrency on performance through architecture analysis of realistic benchmark problems and comparison with measured performance.

Understanding the impact of architecture concurrency on the performance of a system, in particular response time and throughput, has the following benefits:

- Traditional performance modeling and analysis can predict the performance of a particular system for a particular hardware configuration. This provides insight into whether the

performance requirements are met. However, it provides only limited insight as to how to improve the performance if the requirements are not met.

- An understanding of the constraints architecture concurrency places on system performance provides insights into potential performance gains that can be achieved purely through an increase in computing resources.
- An understanding of the concurrency characteristics of different architecture patterns provides the basis for systematic architecture changes that will result in predictable improvements of system performance.

3.2 Background

Embedded computer systems have evolved from small software systems with monolithic structures and no operating system that execute on special hardware, to large-scale systems composed of concurrent applications executing on a common set of distributed computing resources. Applications embodying static execution timelines and interaction through shared data areas are being replaced by preemptive scheduling and predictive scheduling analysis to improve resource utilization and ease system evolution. However, without concurrency control concurrent execution resulting from preemptive scheduling and distributed processing can introduce computational and communication non-determinism, resulting in increased potential for timing and data precedence faults. Many of the detrimental effects are not realized until integration, and sometimes not until system fielding. This problem is compounded as multiple signal streams are processed in a common set of computing resources, and signal data from different sensors are integrated to improve the operation and performance of the system being controlled. Embedded computer systems have evolved from providing basic control capability to systems that include capabilities and processing requirements similar to those found in command-and-control systems. Such an evolution can be witnessed in airplanes, automotive systems, and autonomous robotics systems.

Concurrency has been of concern to the research community for many decades. In the seventies and eighties models for validating the use of synchronization mechanisms to control concurrency without deadlock or starvation were investigated. Examples include consistent use of semaphores as synchronization primitives [Habermann 67], concurrent sequential processes as formalized abstraction [Hoare 85], and Rendezvous as programming language abstraction [Ada 83]. In the eighties and nineties, prediction of schedulability and the impact of shared resource locking were addressed, for example through rate monotonic analysis [Klein 93]. More recently, investigators have studied abstract concurrency models and validation of application code against these abstractions, for example in the Fluid project led by Scherlis at Carnegie Mellon University [Greenhouse 2003].

Research in multi-processor and network systems has examined performance prediction through queuing networks applied to large-scale multi-processor systems [Fuller 78, Jones 78]; distributed resource allocation under timing, load balancing, and fault tolerance constraints, for example Carnegie Mellon's work on TimeWeaver, applied to several industrial

partner project in the Defense Advanced Research Projects Agency (DARPA) Model-Based Integration of Embedded Software (MoBIES) program [Rajkumar 2003, DeNiz 2004]; and dynamic load adaptation for critical system processing flows to adjust to overload conditions, investigated for command-and-control systems such as those found in the DD(X) destroyer [Welch 98].

As mentioned, performance modeling research has its roots in performance prediction for multi-processor systems and networks through queuing models. Applied to software systems, performance modeling takes the form of methods based on queuing networks, process algebras, Petri-nets, simulation, trace analysis, and stochastic processes. Balsamo provides an excellent survey of the state of performance prediction [Balsamo 2004]. This survey shows that only recently have attempts been made to base performance modeling on architecture descriptions.

3.3 Approach

Our approach in developing an analytical framework for determining the impact of architecture concurrency on performance measures of latency and throughput is based on two actual system architectures and consists of three parts:

1. identification of key architecture characteristics and performance implications for each of the two actual customer systems through our external collaborators
2. determination of the benefits and limitations of deploying an architecture optimization technique from one of the customer systems to another embedded system architecture
3. development of an analytical method for predicting latency and throughput for flow-based embedded system architecture in the context of architecture concurrency

3.4 Collaborators

The SEI team consisted of Peter Feiler, Aaron Greenhouse, Jorgen Hansson, John Hudak, and Lutz Wrage, and had two external collaborators.

The first external collaboration involved Dave Statezni from Rockwell Collins through his own support. Through this collaboration the team had access to an avionics system consisting of 15 subsystems operating in networked multiple processing nodes. Through collaboration prior to this project we had performed a pattern-based analysis of this avionics architecture and our collaborator had produced a machine-processable architecture model of an avionics system in the industry-standard Society of Automotive Engineers (SAE) Architecture Analysis and Design Language (AADL) notation. This model includes timing data and flow specifications of key signal flows. This benchmark is representative of control system applications that have moved from static timeline scheduling to preemptive scheduling. In addition, Rockwell Collins has migrated to a partitioned runtime architecture according to the ARINC653 standard.

A second external collaboration involved Michael Moore from the South West Research Institute (SwRI) through his own support. He provided us with information and documentation of a system that was a challenge problem in the DARPA MoBIES program. This system is representative of applications that require operating in a large classification search space to process data sets. An initial architecture model of this message classification and feature identification system has been created and a multiple processor networked system implementation has been constructed using TimeWeaver technology from Carnegie Mellon's IMAGES project (led by Raj Rajkumar). SwRI has collected performance measurements demonstrating throughput and latency improvement. Performance gain in this system was achieved in two ways. First, an architecture optimization called common subgraph elimination was applied to remove redundant processing steps. Second, a multi-processor hardware architecture was used to improve performance through concurrent processing.

3.5 Evaluation Criteria

We have proposed that this project be evaluated based on our success in achieving the following:

1. identification of architecture concurrency abstractions that are relevant to predicting latency and throughput of flow-based system architectures
2. development of an algorithm for flow latency analysis in the context of the Open Source AADL Tool Environment (OSATE) for SAE AADL, its application to the avionics system architecture model, and its validation through performance data from the signal classification system
3. development of an analytical method for predicting throughput under the concurrency abstractions identified in (1), which can be represented in SAE AADL, and validation of the method by performance data from the customer system
4. identification of the benefits and limitations of common subgraph elimination as an architecture optimization supported by performance measurements

Successful demonstration of the analytical methods for predicting and improving throughput and latency in the context of SAE AADL-based architecture models has the potential to become an engineering tool in the Model-based Engineering for Embedded Systems effort of the Performance-Critical Systems initiative in the Dynamic Systems program of the SEI. This would impact a number of SEI customers as they start to embrace the SAE AADL as an industry standard.

3.6 Results

Examination of the embedded system architectures from our two collaborators identified both application architectures as flow-based architectures that can be modeled through a pipeline pattern. By mapping this pipeline pattern into a model expressed in SAE AADL notation, we

were able to leverage the task and communication abstractions of AADL with precisely defined semantics for execution and communication as the basis for an analytical framework for latency and throughput analysis.

Furthermore, we were able to leverage an architecture analysis case study on the avionics system [Feiler2004] that was performed prior to the start of this project. This case study gave us initial insight on the importance of well-defined communication timing semantics to identify potential latency issues when migrating from a cyclic executive to a preemptively scheduled system and to a partitioned runtime architecture. Our insights regarding the change in timing characteristics of tasks and communication for such a system has been included as content material for an SEI public course offering titled Model-Based Engineering for Embedded Systems with AADL.

The examination of the signal classification system allowed us to identify key architecture patterns that made common sub-expression elimination a successful tool for optimizing throughput. The available measurements showing performance improvement through the use of this optimization technique led us to investigate whether this optimization technique has broader applicability.

Based on the above findings we developed an analytical method for predicting flow latency based on task execution and communication timing characteristics. This algorithm demonstrates that sampled processing and sampled communication constrains any reduction in flow latency that can be gained through additional processors. In other words, latency is strongly affected by the sampled processing and delayed communication characteristics of a pipeline pattern, and can primarily be improved by changing the application pattern itself. We have investigated the use of this analytic method for making lower bound predictions for high-level architecture models of partitioned systems, and for detailed models of architectures with a thread-based concurrent execution architecture and mid-frame and phase-delayed communication characteristics. We have implemented this analytic method as an algorithmic analysis plug-in for the Open Source AADL Tool Environment. We have applied this analysis plug-in to the 20,000-line AADL model of the avionics system architecture to identify potential latency issues. Performance measurements from the signal classification system architecture provide a validation of the predictive nature of our analytical model in that there is little impact of additional processors on the latency.

We are also developing an analytic method for predicting throughput and throughput improvements through additional processors. In this case we had to redefine the concept of throughput for systems that perform data sampling, such as control systems, in order to be able to predict performance improvement. This approach allows us to develop an algorithm that can handle both pipelines with complete processing message sequences and processing of sampled data streams. Performance data from the collaborators provides a validation of this analytic approach as well. An algorithmic implementation of this analysis method may not be completed until after the completion of this project.

Finally, investigation of the common subgraph elimination technique has identified the effectiveness of flow fan-out as a key characteristic in a system architecture. We have identified application areas and development processes that result in placation architectures with these fan-out characteristics. SAE AADL has explicit support for modeling flows through the runtime architecture of embedded systems and allows us to do so at different levels of the system component hierarchy. The degree of fan-out affects the benefit of this optimization. A second factor is the depth to which the different flow paths have common processing steps. In the context of this optimization commonality of components must be carefully defined to include not only the component implementation being the same, but also any calibration parameter values used to tune the application from a domain perspective. An example is calibration of a controller or filter. A limiting factor of this optimization is that its application to a fault tolerant system architecture can potentially undo the intended redundancy. For a replication-based redundancy scheme this optimization would remove any intended fan-out to redundant components. Thus, an application architecture must be appropriately annotated to ensure that this optimization is not applied under those circumstances.

3.7 Course and Publications

"Model-Based Engineering with AADL," public course offering by Software Engineering Institute; includes sessions on Flow Latency Analysis in High-Level Model, and on Latency Impact of Migrating to Preemptively Scheduling & Partitioned Thread Architecture.

Impact of Architecture Concurrency on Latency and Throughput, Software Engineering Institute, Technical Note, forthcoming.

Common Subgraph Elimination as Architecture Optimization, Software Engineering Institute, Technical Note, forthcoming.

"Open Source AADL Tool Environment (OSATE) Release 1.1," includes prototype of the Flow Latency Analysis Plug-in, available at http://www.aadl.info.

3.8 References

URLs are valid as of the publication date of this document.

[Ada 83] Ada 83 Reference Manual. Available at http://www.adahome.com/Resources/refs/83.html.

[Balsamo 04] Balsamo, S.; Di Marco, A.; Inverardi, P.; & Simeon, M. "Model-Based Performance Prediction in Software Development: A Survey." *IEEE Transactions on Software Engineering*, May 2004.

[De Niz 04] De Niz, D. “Modeling Functional and Para-Functional Concerns In Embedded Real-Time Systems.” PhD diss., Carnegie Mellon University, Electrical & Computer Engineering, April 2004.

[Feiler 04] Feiler, P.; Gluch, D.; Hudak, J.; & Lewis, B. *Embedded Systems Architecture Analysis Using SAE AADL* (CMU/SEI-2004-TN-005). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.

[Fuller 78] Fuller, S.; Ousterhout, J.; Raskin, L.; Rubinfeld, P.; Sindhu, P.; & Swan, R. “Multi-Microprocessors: An Overview and Working Example.” *Proceedings of the IEEE*, February 1978.

[Greenhouse 03] Greenhouse, Aaron; Halloran, T.J.; & Scherlis, William L. “Using Eclipse to Demonstrate Positive Static Assurance of Java Program Concurrency Design Intent.” Eclipse Technology eXchange (eTX) Workshop, *2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Available at http://doi.acm.org/10.1145/965660.965681.

[Habermann 67] Habermann, A.N. “On the Harmonious Cooperation of Abstract Machines.”, PhD thesis, Technological University Eindhoven, Department of Mathematics, 1967.

[Hoare 85] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice-Hall Publishers, 1985.

[Jones 78] Jones, A.; Chansler, R.; Durham, I.; Feiler, P.; Scelza, D.; Schwan K.; & Vegdahl, S. “Programming Issues Raised by a Multiprocessor.” *Proceedings of the IEEE*, February 1978.

[Klein 93] Klein, M.; Ralya, T.; Pollak, B.; Obenza, R; & Gonzalez Harbour, M. *A Practitioner’s Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[Rajkumar 03] Rajkumar, R. & de Niz, D. “TimeWeaver: A Software-Through-Models Framework for Real-Time Systems.” *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’03)*. San Diego, CA, June 11-13, 2003. ACM 2003, ISBN 1-58113-647-1.

[Welch 98] Welch, L.; Shirazi, B.; Ravindran, B.; & Bruggeman, C. “Specification and Modeling Of Dynamic, Distributed Real-time Systems.” *Proceedings of the 19th IEEE Real-Time Systems Symposium*. Madrid, Spain, 2-4 December, 1998. IEEE Computer Society Press, 1998. Available at http: //computer.org/proceedings/rtss/9212 /9212toc.htm.

4 The Impact of Function Extraction Technology on Next-Generation Software Engineering

Richard C. Linger, Mark G. Pleszkoch, Stacy J. Prowell, Gwen Walton, Alan R. Hevner, Rosann W. Collins

4.1 Purpose

Traditional engineering disciplines depend on rigorous methods to evaluate the expressions (e.g., equations) that represent and manipulate their subject matter. Yet current-generation software engineering has no practical means to fully evaluate the expressions it produces. In this case, the expressions include software designs and computer programs, and evaluation means understanding their full behavior, right or wrong, intended or malicious. Short of an impractical expenditure of resources, no programmer can say for sure what the behavior of a sizable program is in all circumstances of use, a reality that lies at the heart of many problems in software. The result of this technology gap is deployment of systems containing unknown errors, vulnerabilities, and malicious code. The risks are substantial for acquisition organizations that lack the means to validate the full behavior of delivered systems, and offshore development of U.S. software further compounds the problem for homeland security.

Current-generation software engineering operates in a world of unknown program behaviors that no amount of effort seems able to surmount. For example, no testing effort, no matter how extensive, can exercise more than a small fraction of possible system behaviors. Lacking better technology, behavior discovery today is a haphazard and imprecise drain on resources carried out by program reading and analysis with full human fallibility. Yet comprehensive knowledge of software behavior is essential for fast development and verification of programs.

While this problem is pervasive today, it need not be so in the future. We believe that a key enabling capability for next-generation software engineering is the transformation of program behavior analysis from an error-prone, resource-intensive process in human time scale into a precise, automated calculation in CPU time scale. An emerging technology termed function extraction (FX) holds promise to make this next-generation capability a reality. The objective of FX technology is routine, automated calculation of the full functional behaviors of programs. The semantics of program behavior revealed by FX methods directly address the DoD challenges of determining expected properties of software systems before they are built, con-

firming their as-built properties, and dramatically decreasing the amount of effort required for implementing new software-intensive systems.

This independent research and development project has addressed the following goals:

- Determine the potential impact of FX technology on software engineering activities from specification and design to implementation and testing.
- Investigate techniques for integrating FX into next-generation software engineering, characterized by fast and correct program development and rapid composition, validation, and evolution of systems.
- Examine the potential for FX technology to become an SEI initiative in the future. In particular, the study focused on the extent to which further investments in FX can serve the needs of SEI clients.

The primary work product of the study was an SEI technical report [Hevner 05] detailing

- the impacts of FX technology on software engineering life-cycle activities
- a recommended approach to implementing FX methods in software engineering tools and practices
- examples of FX applications in various software engineering contexts
- evaluation and feedback from a major software organization on the impact of FX technology on its software engineering capabilities

The need for this study was exemplified in a recent analysis by the National Institute of Standards and Technology that reports that faulty software costs the U.S. economy nearly $60 billion annually in breakdowns and repairs. FX technology has the potential to improve the practice of software engineering to help reduce such waste and inefficiency.

4.2 Background

The study of function extraction was initiated in the SEI CERT Program, resulting in publication of a paper detailing the technology and its potential [Pleszkoch 04], and development of a proof-of-concept prototype. This work led to sponsorship of an ongoing CERT project to develop the Function Extraction for Malicious Code (FX/MC) system.

4.2.1 The Idea of Function Extraction

Function extraction deals with the semantics of software behavior. All levels of abstraction in the development of software systems deal with behavioral semantics, from low-level machine language operations to high-level system capabilities. As software systems are developed and evolve over time, semantic content is continuously created, intentionally or unintentionally, correct or incorrect. Effective development and evolution of a system depends on how well its behavior is understood by its developers. The complexity and quantity of semantic infor-

mation can overwhelm developers, leading to loss of intellectual control. This loss of semantic understanding occurs for many reasons at all levels of a system. **Error! Reference source not found.** illustrates examples of the creation and inevitable loss of behavioral semantics information, from individual chips to entire information systems.

The ultimate goal of function extraction is to calculate full semantic behavior at all levels of system abstraction, from specification to design to implementation. This goal can be achieved by automating the computation and composition of behaviors in the languages employed to express such artifacts. These languages, whatever their level of abstraction, embody definitions of the behavioral semantics of their structures and rules of combination. These semantics can be captured and employed for function extraction as shown in **Error! Reference source not found.**.

The function extraction process at any system level begins with a well-defined language whose semantics can be captured in terms of the functions of language structures and the rules that govern their combination. An automated function extractor can then be developed for the language. Any system artifact written in that language can then be submitted to the function extractor, which can produce a behavior catalog containing all the behavior defined by the artifact.

Level	Creation and Loss of Semantic Knowledge
Processors	**Creation:** engineers create the behavioral semantics of chip operations by combining circuits **Loss:** errors and ambiguities in processor manuals
Languages	**Creation:** designers create the behavioral semantics of language instructions by combining chip operations **Loss:** errors and ambiguities in language manuals; compilers define semantics
Components	**Creation:** programmers create the behavioral semantics of components by combining language instructions **Loss:** full functional behavior of components not documented
Applications	**Creation:** programmers create the behavioral semantics of applications by combining components **Loss:** “Bob knows the application, but he’s retiring.”
Systems	**Creation:** engineers create the behavioral semantics of systems by combining applications **Loss:** systems “go natural” from accumulated knowledge loss

Table 4-1: Creation and Loss of Semantic Information in Software Development

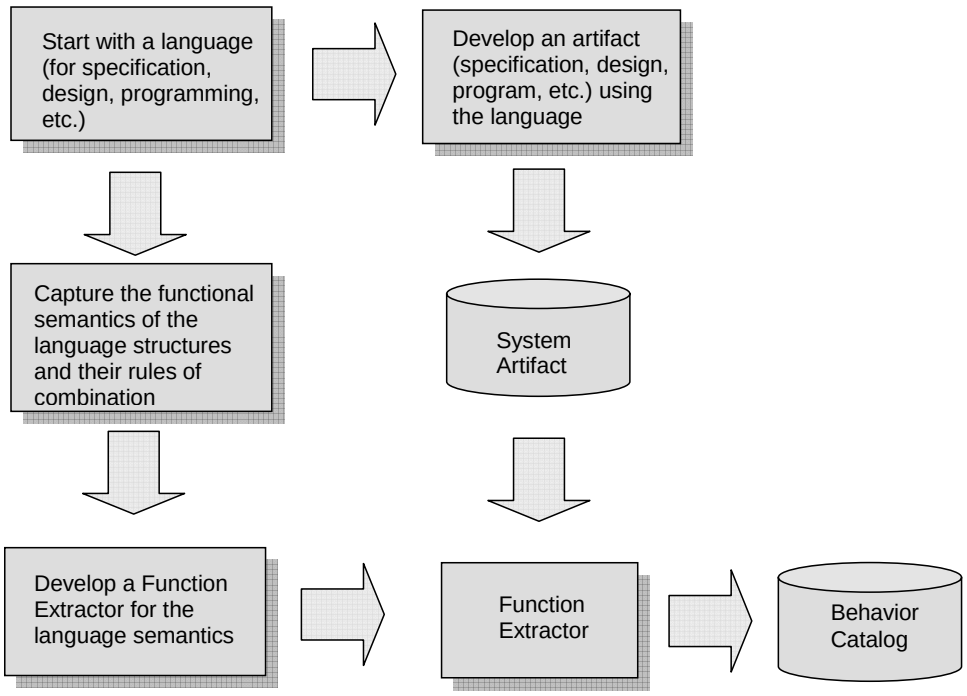

Figure 4-1: The Basic Concept of Function Extraction

4.2.2 Fundamentals of Program Behavior Calculation

The function-theoretic model of software [Hausler 90, Hevner 02, Hoffman 01, Linger 79, McCarthy 63, Mills 86, Mills 02, Pleszkoch 90, Prowell 99] treats programs as rules for mathematical functions. The purpose of automated behavior calculation is to extract the full functional behavior of programs, that is, how programs transform inputs into outputs in every circumstance, and present the behaviors to users as precise as-built specifications in procedure-free form for analysis.

The fundamental insight in function extraction technology is the realization that, while sizable programs contain a virtually infinite number of execution paths, they are constructed of a finite number of nested and sequenced control structures, each of which makes a finite contribution to overall behavior. These structures correspond to mathematical functions or relations, that is, mappings from inputs to outputs. The functional mappings can be automatically extracted in a stepwise process that traverses the finite control structure hierarchy. At each step, details of local code and data are abstracted out, while their net effects are preserved and propagated in the extracted behavior. While no general theory for loop abstraction can exist, use of recursive expressions and patterns for loops provides an engineering solution. The mathematical foundations for function-theoretic behavior calculation are currently being applied to the specialized problem of malicious code analysis in the Function Extraction for

Malicious Code project, which is being developed to determine the behavior of malicious code expressed in the Intel Assembler Language. Based on this initial experience, the purpose of this independent research and development project was to explore the full effect that FX technology can have on the broader software engineering life cycle.

4.3 Approach

In order to inform and guide the direction of FX research and development, our research focused on better understanding how enterprises can employ FX technologies in their software engineering environments. **Error! Reference source not found.** shows a portion of the opportunities we have studied to evaluate next steps for FX evolution. It lists software development activities in the rows and potential language environments in the columns. The highlighted cell is our starting point—the current project to develop a function extractor for Assembler Language. The table also includes several FX-related tools, for example, structure transformers to extract the control flow behavior of programs and express it in structured form, component composition generators to calculate the net behavior of composed components, and behavior catalog analyzers to respond to user queries on behavioral properties of interest. To help determine the next steps of development, we have gathered and analyzed data from potential users of FX technologies. With this information, we are able to recommend high-pay-off areas for FX research and development.

To structure the research study, we posed two sets of three questions each. It is generally understood that program comprehension is a critical aspect of all software development and maintenance activities [Rajlich 02]. Prior research has found that both program and task characteristics interact to impact the nature of program comprehension [Storey 99], thus it is important to develop tools with specific software engineering activities in mind. Therefore, the first three research questions for this study focused on understanding current approaches to, cost of, and impacts of program comprehension, with particular attention paid to how these vary by type of activity:

Research Question 1: What techniques are in current practice to understand and document program behavior?

Research Question 2: What are the typical costs of program comprehension and documentation to development?

Research Question 3: What is the relationship of program comprehension and system quality?

Life Cycle Activity	Specification Automation	Architecture Automation	Assembler Automation	C Automation	C++ Automation	Java Automation	Other Lang. _____
Specification Development	**Specification Behavior Extractor** **Behavior Catalog Analyzer**						
Architecture Development		**Architecture Behavior Extractor** **Behavior Catalog Analyzer**					
Component Development: Evaluation & Selection and Design & Implementation			**Structure Transformer** **Function Extractor** **Behavior Catalog Analyzer**	**Structure Transformer** **Function Extractor** **Behavior Catalog Analyzer**	**Structure Transformer** **Function Extractor** **Behavior Catalog Analyzer**	**Structure Transformer** **Function Extractor** **Behavior Catalog Analyzer**	**Structure Transformer** **Function Extractor** **Behavior Catalog Analyzer**
Correctness Verification			**Correctness Verifier**	**Correctness Verifier**	**Correctness Verifier**	**Correctness Verifier**	**Correctness Verifier**
System Integration			**Component Composition Generator** **Behavior Catalog Analyzer**	**Component Composition Generator** **Behavior Catalog Analyzer**	**Component Composition Generator** **Behavior Catalog Analyzer**	**Component Composition Generator** **Behavior Catalog Analyzer**	**Component Composition Generator** **Behavior Catalog Analyzer**
System Testing			**Behavior Catalog Analyzer**	**Behavior Catalog Analyzer**	**Behavior Catalog Analyzer**	**Behavior Catalog Analyzer**	**Behavior Catalog Analyzer**
System Maintenance and Evolution			**Behavior Catalog Analyzer**	**Behavior Catalog Analyzer**	**Behavior Catalog Analyzer**	**Behavior Catalog Analyzer**	**Behavior Catalog Analyzer**

Table 4-2: FX Impacts—Where to Next?

The second set of research questions centered on views about the potential of FX from developers who have knowledge of the technology:

Research Question 4: In which system development activities and environments does FX technology have the potential for greatest impacts?

Research Question 5: What are the potential impacts of FX technology on other software engineering technologies and issues?

Research Question 6: What are the challenges to adoption of FX technology?

A carefully designed empirical study was performed to answer these research questions. Study participants were experienced system developers. The study questionnaire was created by the project research team and was pilot tested with an academic audience composed of professors and graduate students at a major research university. The study was then conducted at a Fortune 100 company with a large and sophisticated group of software developers. The session began with a presentation on FX technology to a roomful of software developers and remotely located developers through a Web cast. The remote group could see the presentation slides and had two-way audio. The training presented function extraction technology and detailed examples of how it could work in software development. This presenta-

tion lasted approximately 90 minutes, followed by a question-and-answer session. After the training session, the researchers asked participants to complete the questionnaire on potential impacts of FX technology in their organization. Software engineers from both on-site and remote locations provided usable questionnaire data. A full description of the results of this study is available in an SEI technical report [Hevner 05]. The data gathered led to the recommendations enumerated in the Results section below.

4.4 Collaborations

The SEI team for this project was composed of Richard Linger, Gwen Walton, Mark Pleszkoch, and Stacy Prowell. Collaborating on the project as SEI Visiting Scientists were Alan Hevner and Rosann Collins from the University of South Florida.

4.5 Evaluation Criteria

This study has produced results that clearly reflect the needs and objectives of SEI sponsors and clients. Specifically, the project has

- specified the impact of function extraction technology on software engineering life cycle activities
- assessed the risks and rewards of investment in FX development
- performed an investigation of the potential for function extaction technology to become a major SEI initiative in the future

4.6 Results

This section identifies next steps for the FX research and development program based on the industry survey data gathered. The data clearly indicated the need for the six project goals listed below. In addition, a seventh goal not discussed in the survey instrument is recommended by the project study authors. Thus, we recommend that the evolution of FX technology be focused on achievement of these goals:

Goal 1: Complete Development of the FX Prototype for Assembler Language Programs

It is important that the FX project continue with development and deployment of the FX/MC system. The Assembler Language environment was rated as the most important for showing FX impacts by the surveyed software engineers. An operational FX system for understanding malicious code will be a key advantage in demonstrating the potential of the technology to industry.

Goal 2: Create FX Automation to Verify Correctness of Programs

The software engineers identified the activity of correctness verification as having the greatest potential for FX impacts. Software developers are demanding improved methods for un-

derstanding the behaviors of programs and verifying the correctness of these behaviors with respect to specifications and designs. This information tells us that a near-term goal of the project must be to demonstrate automation of program correctness verification using FX technology.

Goal 3: Create FX Automation for High-Level Programming Environments Starting with Java

The software engineers rated the programming languages Java, C, and C++ as very important for the application of FX technology. It is clear that the software development industry has great need for support in understanding the behaviors of programs written in these high-level languages. Thus, another important near-term goal of the FX project will be to develop a function extractor prototype for one or more of the most popular programming languages. The engineers recommended Java as their first choice.

Goal 4: Perform Research on Semantics of System Specification and Architecture for FX Automation

The software engineers in the survey demonstrated concern and even skepticism that the promise of FX theory can be successfully transitioned into effective engineering practices for the front-end activities of system specification and architecture development. The reason is the inability with state-of-the-art methods to rigorously define and represent the semantics of software specifications, architectures, and high-level designs. In fact, FX technology is seen as having little near-term impact on these early life cycle activities due largely to the lack of well-defined semantics in these areas. An initiative to perform research on the semantics of software system specification and architecture is required for FX technology to be applied in these activities.

Goal 5: Perform Research on Human/Computer Interfaces for FX Automation

Effective use of innovative technologies such as FX depends on adaptable and user-friendly human/computer interfaces. It is important that research on user interfaces for FX be performed in parallel with development of the automation itself. Computed program behavior has not been available to software engineers in the past, and new reasoning and analysis patterns are sure to emerge. Research is required to understand the dynamics of this new augmentation of human intelligence for optimal design of its user interfaces.

Goal 6: Perform Experimentation with FX Technology to Evaluate its Impact

Scientific research requires rigorous experimentation to evaluate the quality and effectiveness of results. The artifacts of FX research and development are theories, practices, and automated tools [Hevner 04]. Empirical evaluation of these artifacts will provide the evidence required by eventual users to accept and adopt FX as an element of their software development processes. Any new technology faces initial resistance because it requires a learning curve and changes in entrenched practices. Rigorous experimentation with FX technologies

resulting in clear evidence that they improve development productivity and system quality will ease their acceptance [Green 04, 05]. **Goal 7: Perform Research on the Semantics of Software Quality Attributes for FX Automation**

In the current state of the art, analysis of software quality attributes such as performance and security is often carried out through subjective, a priori evaluations that provide little value in the dynamics of system operation, where attribute values can change quickly. A capability to compute quality attribute values with mathematical precision will permit both rigorous assessment and improvement of attributes during software development, and the real-time evaluation of system attributes during operation. Research is required to define computational models for quality attributes that can be evaluated by FX automation. That is, quality attributes must be treated as functions to be computed as dynamic properties of systems.

In summary, the findings of this study define a rich program of research and development in FX technology that can make a major contribution to next-generation software engineering. Guided by these findings, the FX team intends to continue development of the FX/MC system, create a correctness verification prototype based on FX technology, and investigate theory and practice for defining software quality attributes in computational terms. In addition, the team will conduct empirical experimentation on reasoning methods and perceptions of FX users to better understand how to apply the technology to augment human capabilities.

4.7 References

URLs are valid as of the publication date of this document.

[Green 04] Green, G.; Collins, R.; & Hevner, A. "Perceived Control and the Diffusion of Software Development Innovations," Journal of High Technology Management Research, Vol. 15, No. 1, February 2004, pp. 123-144.

[Green 05] Green, G.; Hevner, A.; & Collins, R. "The Impacts of Quality and Productivity Perceptions on the Use of Software Process Improvement Innovations." Information and Software Technology, Vol. 47, No. 8, June 2005, pp. 543-553.

[Hausler 90] Hausler, P.; Pleszkoch, M.; Linger, R.; & Hevner, A. "Using Function Abstraction to Understand Program Behavior." *IEEE Software, Vol. 7*, No. 1, IEEE Computer Society Press, Los Alimitos CA, January 1990.

[Hevner 02] Hevner, A.; Linger, R.; Sobel, A.; & Walton, G. "The Flow-Service-Quality Framework: Unified Engineering for Large-Scale, Adaptive Systems." *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, Hawaii, January 2002. IEEE Computer Society Press, Los Alamitos, CA, 2002.

[Hevner 04] Hevner, A.; March, S.; Park, J.; & Ram, S. "Design Science Research in Information Systems." *Management Information Systems Quarterly, Vol. 28*, No. 1, March 2004, pp. 75-105.

[Hevner 05] Hevner, A.; Linger, R.; Collins, R.; Pleszkoch, M.; Prowell, S.; & Walton, G. *The Impact of Function Extraction Technology on Next-Generation Software Engineering* (CMU/SEI-2005-TR-015). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, July 2005.

[Hoffman 01] Hoffman, D. & Weiss, D. (eds.). *Software Fundamentals: Collected Papers by David L. Parnas*. Addison Wesley, Upper Saddle River, NJ, 2001.

[Linger 79] Linger, R.; Mills, H.; & Witt, B. *Structured Programming: Theory and Practice*. Addison Wesley, Reading, MA, 1979.

[McCarthy 63] McCarthy, J. "A Basis for a Mathematical Theory of Computation." *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, eds. North-Holland, Amsterdam, 1963.

[Mills 86] Mills, H.; Linger, R.; & Hevner, A. *Principles of Information System Analysis and Design*. Academic Press, San Diego, CA, 1986.

[Mills 02] Mills, H. & Linger, R. "Cleanroom Software Engineering." *Encyclopedia of Software Engineering, 2nd ed.*, J. Marciniak, ed. John Wiley & Sons, New York, 2002.

[Pleszkoch 90] Pleszkoch, M.; Hausler, P.; Hevner, A.; & Linger, R. "Function-Theoretic Principles of Program Understanding." *Proceedings of the 23rd Annual Hawaii International Conference on System Science*. Hawaii, January 1990. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Pleszkoch 04] Pleszkoch, M. & Linger, R. "Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior." *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*. Hawaii, January 2004. IEEE Computer Society Press, Los Alamitos, CA, 2004.

[Prowell 99] Prowell, S.; Trammell, C.; Linger, R.; & Poore, J. *Cleanroom Software Engineering: Technology and Practice*. Addison Wesley, Reading, MA, 1999.

[Rajlich 02] Rajlich, V. & Wilde, N. "The Role of Concepts in Program Comprehension." *Proceedings of the 10th International Workshop on Program Comprehension (IWPC '02)*. IEEE Computer Society Press, 2002.

[Storey 99] Storey, M.A.D.; Fracchia, F.D.; & Muller, H.A. "Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration." *The Journal of Systems and Software, Vol. 44*, No. 3, January 1999.

5 Issues in Scalability

Charles Weinstock, John Goodenough

5.1 Purpose

In 2003 teams from three federally funded research and development centers (FFRDCs)—the SEI, the MITRE Corporation, and the Institute for Defense Analyses—conducted a joint study on software producibility for the Undersecretary of Defense for Science and Technology. Software producibility was defined as the ability to deliver software-based capability predictably and efficiently.

The impetus for forming this study group was the increasing perception in government and elsewhere that software producibility is a problem. Edward C. "Pete" Aldridge, the former Under Secretary of Defense for Acquisition, Technology and Logistics, was quoted as saying: "[Software] continues to grow in importance in our weapons systems—and remains a significant contributor to program cost, schedule and performance shortfalls."[1] Delores Etter, currently Assistant Secretary (Research, Development & Acquisition) for the U.S. Navy, said in 1999[2] that half of all DoD software projects cost more than double their initial cost estimates, projects slipped an average of 36 months, and a third of all projects ended up being canceled.

One of the key findings of the FFRDC team was that although management issues are significant contributors to most DoD program shortfalls, there are also clear underlying science and technology gaps. These technical problems are often only clearly revealed when the systems are integrated. The problem is that the technical basis for software producibility is deficient. We lack the engineering basis to accurately predict behavior, performance, and other key properties of large and complex software systems before they are built.

One clear example of our lack of technical basis for predicting system behavior is illustrated by the fact that a significant number of systems fail in initial use (or even in integration) because they do not scale well, that is, factors that have a negligible effect when systems are below a certain scale (on some dimension) have a harmful effect as the scale increases. Understanding the effects of scale on a system is the scalability problem. Although the problem is not new, the increasing size of DoD systems (e.g., increasing code size, increasing number

[1] Testimony of the Undersecretary of Defense (Acquisition, Logistics, and Technology) to the House Armed Services Committee, July 12, 2001, armedservices.house.gov/ openingstatementsandpressreleases/107thcongress/01-07-12aldridge.html

[2] Etter, D. Acquisition Software Oversight, Crosstalk, August 1999, p. 3, (http://www.stsc.hill.af.mil/crosstalk /1999/08/etter.pdf)

of users, increasing scope of demands) makes the problem more critical today than in the past. And although increase in size usually creates management challenges, technical issues also arise.

The purpose of this study was to look more closely at the scalability problem. We wanted to characterize technical sources of scalability problems, existing solutions, and technical gaps. Given a better understanding of scalability issues, we hoped to make it possible for engineers to identify potential scaling problems earlier in system design and develop better solutions. A good understanding of scalability issues could also help ensure improved acquisition outcomes.

5.2 Background

When we proposed this independent research and development project, we thought we knew what the attribute we call "scalability" was. We expected to find a significant literature on the subject and hoped to be able to apply techniques that we've been developing at the SEI, such as assurance case technology[3] to some of the identified problems.

As we delved more and more into the literature surrounding scalability we found that this was not actually the case. There were a lot of papers that purported to deal with the scalability problem, but we were hard pressed to find any papers with a crisp definition of the term, much less a categorization of or solution to the problem of developing large systems so that they become scalable. Indeed, we discovered that we didn't fully understand or appreciate the meaning of the term ourselves. It turns out that everyone "knows" what scalability is until they start to dig deeper and try to state underlying principles and concerns.

5.3 Approach

Our initial approach was to conduct a literature survey on the subject and to interview colleagues at the MITRE Corporation who had participated in programs that had experienced scalability problems. The collaboration with MITRE was intended to help us identify scalability issues, explain their consequences in the particular systems studied, analyze effective and ineffective approaches for identifying and solving these issues, and develop proposed standard analysis patterns for identifying the issues and evaluating potential solutions.

As a result of the literature search, we had hoped to identify technology gaps and relate these gaps to promising approaches. However, as already mentioned, the results of our literature survey were not at the level we expected, and as we describe below the collaboration with MITRE failed to get off the ground despite the best intentions of both parties.

[3] See CMU/SEI-2004-TN-016, available at http://www.sei.cmu.edu/publications/documents/04.reports/04tn016.html.

In lieu of talking with MITRE engineers, we interviewed several colleagues at Carnegie Mellon's School of Computer Science who had encountered significant real-world scalability problems in both commercial and academic systems. We synthesized the results of these interviews and the issues revealed by the literature survey into an SEI technical note that will be published later in 2005.

5.4 Collaborations

The SEI technical staff members involved on this project are John Goodenough and Charles B. Weinstock. We originally expected to have collaborators from the MITRE Corporation, but this did not come to pass due to unavoidable new commitments on their part and confidentiality concerns that arose about discussing scalability failures observed in their clients' systems,. Nonetheless, we did conduct some very helpful interviews with individuals in the Carnegie Mellon School of Computer Science, including professors Bruce Maggs, Mahadev Satyanarayanan, and David Farber. They volunteered the time required for the interviews.

5.5 Evaluation Criteria

At the onset of this project we specified the following success criteria:

- identification of technical gaps that appear to have a significant impact on the ability of engineers to identify and/or address scalability problems early in system development
- completion of a literature survey that makes clear whether there are unexploited technologies that could improve our ability to engineer systems that scale well
- identification of analysis patterns and approaches that make it easier for engineers and acquisition agents to reliably determine if scalability problems are likely to be critical for a system under development

5.6 Results

Our inability to work with actual systems experiencing scalability problems caused considerable delays for this effort. Although we have accomplished much in surveying the literature and in identifying technical gaps (the first two success criteria above), we have not had an opportunity to begin to explore patterns for analyzing scalability problems.

The current definitions of scalability are rather informal and ad hoc. They deal with symptoms rather than causes—i.e., if a system performs well as load increases, it is considered "scalable." With few exceptions, papers that use the term scalability talk about how performance is better in scalable systems, but not much about principles. The mental model for scalability is fuzzy, and this limits the community's ability to ensure that systems can readily adapt as demand (on some dimension) increases.

In general, with exceptions for parallel computation and distributed simulations, there is no existing framework for understanding the nature of scalability issues. Without such a framework—a "theory of scalability" if you will—it is difficult to put in context the lessons learned on particular systems so future systems can address certain types of scalability problems in a more cost-effective manner. Although engineers can deal with scaling issues as they arise, the overall state-of-the-practice seems to be rather ad hoc. There is a need for a conceptual framework addressing issues of scalability. Such a framework could guide engineers so they could identify potential scalability problems in advance.

In short, we lack a precise characterization of what scalability is, how to measure it, and what factors affect it.

Our literature survey did identify some scalability prediction strategies that may have an impact on the ability to engineer scalable systems. We have also identified a glimmer of a conceptual framework—a key unifying notion that has not been addressed by the literature in other than an ad hoc manner. This is the notion that various dimensions of scalability exist and the technical approaches for each dimension, and for different portions of a given dimension, can differ quite a bit. In our tentative conclusions, we consider a scalability dimension to be a type of system resource; limitations on the resource make systems unusable as load (on the resource) increases. Among the dimensions we have tentatively identified are the usual ones of network bandwidth and CPU capability. Other, less commonly recognized dimensions include

- the administrative dimension—the human support activities needed to keep users happy;
- the human interface dimension—controlling display real estate or the amount and rate of information presented to users;
- the coupling dimension—how modifying a system to increase its capability on one dimension may increase the load on another dimension, which then needs additional modification;
- the business case dimension—how much should be invested up front to improve a system's scalability on one of the other dimensions;
- a modifiability dimension—the ability to retain conceptual control over a system as it evolves; failure on this dimension is exemplified by the situation where the defects in a system reach a steady, and unsatisfactory, level because each bug fix introduces new defects;
- and (perhaps) an openness dimension—decisions that make it easier or more difficult to attract new and varied users to a system; this affects the ability of a system to stimulate higher demand from its potential user domain.

5.6.1 What is Scalability?

System scalability is not a Boolean attribute taking on a true or false answer. All systems scale to some point. Alternatively no system is infinitely scalable. Ultimately, even a system

designed for maximal scalability will run up against physical limitations (e.g., the speed of light.) When designing a system we make informed guesses based, presumably, on requirements, and experience as to the needed resources and the demands on those resources. If we are doing a good job of system design we try to make informed guesses as to the changes that the future will bring—both in terms of resources and in terms of usage patterns. We then try to design the system so that it can accommodate those anticipated changes—perhaps by having all the resources we need from the beginning, or perhaps by designing the system so that resources can be added dynamically as needed. If we're doing a really good job of system design we add some margin to allow for even more significant changes than we forecast. Of course a poor design won't even have taken current resources and usage profiles into account. This happens more than it should and was one of the initial motivations for this study.

5.6.2 Achieving Scalability Involves Making Tradeoffs

Building a system to be scalable almost always requires making tradeoffs with other attributes of the system. In order to achieve ever higher levels of operation, it may be necessary to either give up performance, usability, or some other attribute, or to pay a big monetary price. The typical tradeoff associated with scalability is trading performance at lower levels of load for the ability to handle larger loads when necessary, but there are other possibilities. For instance:

- Designing a system to scale may entail upfront costs that are higher than those of a non-scalable version of the same system.
- It may be necessary to give up fine-grain human control of the system to achieve levels of scalability that would otherwise be unmanageable.
- It may be necessary to design the system with less functionality than would be the case if it did not have to scale.

5.7 Conclusion

Although this project did not achieve one of its key goals—namely the identification of analysis patterns and approaches that make it easier for engineers and acquisition agents to reliably determine if scalability problems are likely to be critical for a system under development—we did identify a key technical gap: the lack of a good framework for understanding the nature of scalability and the need to focus subsequent work toward analysis and prediction.

We believe that we have made significant progress in defining what scalability is and what it is not while laying out the problem space for the further exploration of scalability issues in subsequent efforts.

5.8 References

URLs are valid as of the publication date of this document.

[Goodenough 05] Goodenough, John & Weinstock, Charles B. *Lessons in Scalability*. Software Engineering Institute, Carnegie Mellon University. Forthcoming, November 2005.

6 Proof-Carrying Code

Kurt C. Wallnau, Sagar Chaki

6.1 Purpose

There is an evident need for mechanisms that enhance our ability to *trust* third-party software. In the current era of plug-and-play, off-the-shelf programs are increasingly available as modules or *components* that can be attached to an existing infrastructure. More often than not, such plug-ins are distributed in machine code or *binary* form. How do we establish that the delivered software is trustworthy—that is, that its execution will do no harm , where the definition of "harm" may vary? This is of course a longstanding question in the software industry. There are a number of techniques used in practice, but they tend to fall within three broad, not mutually exclusive, approaches:

1. *Trusted certifier*. In this approach, a trusted third party tests a software product for compliance to specified criteria. A well known example is the National Security Agency's trusted computer security evaluation criteria (TCSEC) and its recent incarnation as the National Information Assurance Partnership (NIAP) Common Criteria and Protection Profiles.
2. *Trusted supplier*. In this approach, software is considered trustworthy if a trustworthy supplier produces it. This is one underlying motive for software process maturity, but cryptographic technology (e.g., digitally signed device drivers) is also seen, tempered by caveat emptor.
3. *Runtime infrastructure*. In this approach, untrusted code will execute within trusted enclaves within a trusted computing base. Enclaves can be established through sandboxing, runtime monitors, or software fault isolation.

These broad approaches have their merits, but suffer obvious limitations. Third-party certification is only as good as the evaluation criteria and the quality of testing. In practice, criteria have substantial subjective content, and testing is expensive and necessarily non-exhaustive. The trusted supplier approach also suffers from subjectivity, but, more fundamentally, from indirectness: nothing is said about any particular software product. Runtime infrastructure has its merits, but incurs additional complexity and a persistent runtime penalty for code execution.

What is needed is an objective, trustworthy, and automated means for establishing proof-level confidence that binary executable code is safe with respect to explicit and unambiguous trust

policies. For this purpose we investigated the maturity of proof-carrying code (PCC), sometimes also referred to as self-certifying binaries.

The objective of this research was to develop an understanding of ongoing research in PCC, to identify its open problems, and to identify areas where transitionable progress has been made. A second objective was to investigate and possibly prototype the use of model checking technology to generate proof certificates of specific safety policies on software delivered from untrusted suppliers.

6.2 Background

In 1997 Necula and Lee demonstrated a new technology, which they called *proof-carrying code* for obtaining objective trust in software [Necula 96, 97]. The basic idea is that the software delivered as binary code contains embedded in the code a proof of its behavior. Generating the proof is expensive, but can be substantially automated; checking the proof is inexpensive and simple, and in fact reduces to a form of type checking at the machine-code level. With PCC, proof-level assurance can be obtained automatically, reliably, and *directly on the executable code itself*.

PCC remains an active area of investigation, with the bulk of the theoretical and practical results produced at the University of California–Berkeley, Princeton, Cornell, Yale, Carnegie Mellon, and recently Harvard. The focus of PCC has generalized somewhat into the topic of certifying compilation, with which it shares many of the same objectives and technologies. Indeed, there is significant overlap in these communities.

6.3 Approach

The research conducted by the SEI sought answers to two questions:

1. What is the general state of PCC research, and what aspects of PCC have neared the state where they can be transitioned into practical use?
2. Can model checking technology developed at Carnegie Mellon and the SEI be adapted to generate proof certificates for certifiably trustworthy code?

To answer these questions we combined literature survey and analysis with practical prototyping.

6.4 Collaborations

This project supported the efforts of one PhD student in the Carnegie Mellon School of Computer Science (SCS), Steven Magill, and one post-doctoral researcher in SCS, Alexsandar Nanevski, now at Harvard. We were also greatly assisted in our research by SCS Professor Peter Lee, a co-inventor of PCC, and SCS Professor Edmund Clarke, co-inventor of model checking.

6.5 Evaluation Criteria

At the outset of the research we established the following success criteria:

- Obtain an accurate description of the current state of the research in PCC.
- Assess the potential for using model checking technology to generate proof certificates.
- Improve collaborative work between the SEI and the Carnegie Mellon School of Computer Science.
- Improve the visibility of the SEI in the computer science research community.

We believe the 2004–05 research satisfied each criterion.

6.6 Results

The research produced three significant results:

- a survey of the state of the research and description of main themes and challenges
- a prototype certifying model checker
- foundations for proving heap and other resource properties with separation logic

6.6.1 Survey of the State of the Research

An online bibliography of more than 70 articles, reports, and tutorial presentations on PCC and related topics was collected. The survey helped to classify the research areas and progress that was outlined in general terms, below. A tutorial presentation of these results was presented to the International Federation of Information Processing Working Group 2.4 (IFIP WG 2.4) on Software Implementation Technology in October.

6.6.1.1 Archetypal PCC

Necula and Lee's work was seminal, and established a characteristic architecture for subsequent research in the field. We refer to this characteristic architecture as archetypal PCC. Its essential elements are depicted in Figure 6-1.

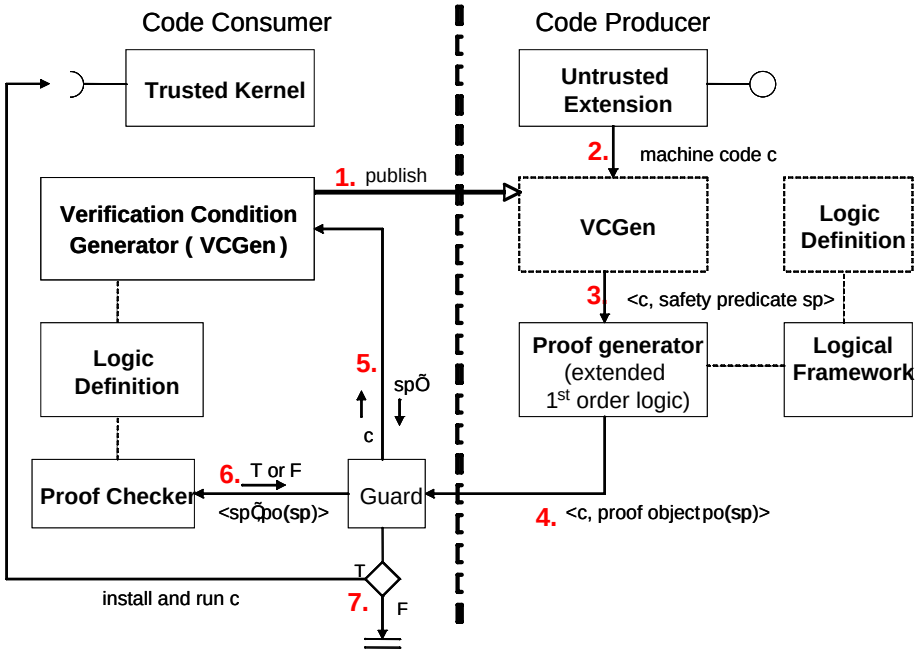

Figure 6-1: Archetypal Proof-Carrying Code

The world is partitioned into code consumers and code producers. An alternative interpretation of this partition that is sometimes useful is that code consumers constitute a "trusted computing base" (TCB), while code producers are untrusted.

The code consumer first publishes a safety policy (number 1 in the figure) that defines precisely what behavioral properties must be satisfied by binary code if it is to be deemed trustworthy. Safety here is a term of art: it refers to a class of behaviors roughly equivalent to "condition X (a bad thing) never happens." The safety policy is specified in terms of a verification condition generator (VCGen, pronounced "vee-see-jen") that defines precisely which proof obligations are generated for each possible (valid) sequence of machine instructions. The verification conditions (VCs) are expressed in standard first-order logic extended with policy-specific terms and rules of inference.

The untrusted producer submits a binary program to a verification condition generator (2 in the figure) that has been developed by the producer to the specifications of the safety policy, or possibly supplied by the consumer. The result is a pair consisting of the binary code and a safety predicate that must be proven true if the code is to be deemed trustworthy.

There are various commercial and academic first-order theorem-proving technologies that can be used to prove the safety predicate (3). For simple safety properties, fully automated proof generation was sometimes possible; for more complex safety properties, in particular those that require the discovery of loop invariants, automation was not always possible, even in principle. *However, it is also important to note that in PCC an important objective is to shift the burden of establishing trustworthiness from the consumer of code to the producer.*

One of the main results of the original Necula and Lee work was to demonstrate an efficient scheme for encoding proofs of the safety predicate. To this end they exploited the famous "propositions as types" symmetry, sometimes known as the Curry-Howard isomorphism. This allows us to define the *type* of a machine code program to be the *proof* of a proposition—the safety predicate. Technicalities aside, archetypal PCC defines an unambiguous encoding of proofs of verification conditions, and enables the specification and development of simple, fast, and efficient proof checkers using mature programming language type-checking techniques.

The code consumer is presented with a binary program and accompanying proof (4). The consumer re-derives the verification condition from the binary code (5), and then checks the validity of the proof versus the code (6). Proof checking is an efficient type-checking algorithm that executes in linear time on the length of the program. Moreover, proof checking is a one-time cost incurred by the code consumer. In contrast, execution monitors, for example, must perform repeated runtime checks of program execution, and sandboxing incurs its own runtime and other costs.

6.6.1.2 Limitations of Archetypal PCC; Related Research

Archetypal PCC established technical feasibility, but also exposed a number of technical challenges that would severely diminish its prospects of widespread technology adoption. For summary purposes these can be thought of as comprising two areas of concern: (1) the size of the trusted computing base; and (2) the expressiveness of certifiable properties and the degree of automation possible in proving these properties. A brief survey of ongoing work will highlight how the research community has been addressing these challenges.

Necula has been an advocate of systematic and incremental improvements to archetypal PCC to improve scale and trust [Schneck 02]. To reduce the size of the TCB, Necula has explored the modularization of safety policies, including techniques by which the untrusted supplier can define safety policies and present proof-level justification that the supplied policy implies (and therefore its proofs will satisfy) the code consumer's published policy [Necula 01a, Schneck 02b]. A related issue to the size of the TCB is the size of encoded proofs in relation to the original binary code; experiments showed that a one-to-three order of magnitude blow-up in the size of the packaged binary could be expected. For small devices or limited bandwidth communication, this could be problematic (granted, these are special cases). Necula has also demonstrated the use of test oracles whereby the consumer must generate rather than check safety proofs, but the untrusted supplier can supply arbitrarily detailed hints on how to *re*-construct the proof [Necula 01].

A more radical PCC agenda called "foundational PCC" has been proposed by Appel and Felty [Appel 00, 01]. In brief, the technical agenda of foundational PCC is to achieve an absolute minimal TCB by developing complete semantic specification of machine code in higher-order (typed lambda) logic. This approach has the virtue of sidestepping difficult issues concerning how the trustworthiness of the safety policies themselves can be established.

In archetypal PCC they are supplied by fiat. In foundational PCC, safety policies are regarded as lemmas to be proven in a more general and primitive, foundational logic; this logic, in turn, relies only on the foundations of mathematics. While there are many technical challenges also posed by foundational PCC, the theory has been demonstrated in many practical application settings by Zhong Shao [Dachuan 04, Zhong 02].

While shrinking the TCB for PCC is essential to realizing the value of proof-level trust in binaries, more immediate and practical concerns for the transition of this technology to practice is the expressiveness of the claims that can be proven about binary code, and the degree to which the construction of proofs can be automated. Significant progress in the area of certifying software model checking offers a promising avenue to make progress on both concerns [Peled 01, Namjoshi 01, Henzinger 02]. Model checking is a fully automated verification procedure that works by exhaustively searching finite models of systems. In some cases, model checkers are able to verify safety properties (something bad *never* happens) and liveness properties (something good *eventually* happens). This offers both the automation and expressiveness needed for transition. Certifying model checkers provide proof certificates as "witnesses" to satisfied properties; conventional model checkers typically only supply witnesses for failures in the form of counterexamples to claims.

6.6.2 Prototype Certifying Model Checker

A software model checking technology was jointly developed by the SEI and Carnegie Mellon for the express purpose of verifying component software [Chaki 04a, Chaki 04b]. This technology uses predicate abstraction and counterexample-guided abstraction refinement to automate the extraction of conservative finite-state models of potentially infinite-state software written in C. Because this technology was designed to model check software directly, rather than indirectly through specialized specification languages (as with most other model checkers), this seemed a good point of departure.

In this research we enhanced the model checker to generate proof certificates of verified safety claims. The novelty of our approach is the use of the Boolean satisfiability (SAT)-solving technology ZCHAFF [Moskewicz 01] to generate proof certificates. Unsatisfiability is the dual of validity. We show that a formula A is valid (i.e., always true) by proving that $\neg A$ is unsatisfiable. ZCHAFF was particularly useful because it generates the "UNSAT core," that is, the subformula whose unsatisfiability ensures the unsatisfiability of $\neg A$ as a whole, and hence proves the validity of A. Hence, only this unsatisfiable subformula is needed to generate the proof certificate, and this is what accounts for the dramatic reduction in the size of the proof certificate. This pioneering use of SAT solving for certifying model checking produced dramatic results in reducing the size of proof certificates up to five orders of magnitude when compared with the proof certificates generated by conventional theorem-proving technology (Table 6-1). Chaki has documented the technical details of these results [Chaki 05].

This result demonstrates a practical enhancement of model-checking technology. Previously, model checkers produced witnesses to failed verification claims ("counterexamples"), but did not produce objective evidence that a claim was satisfied. By providing a proof certificate of the verified claim, we effectively remove the model checker—itself a complex piece of software—from the trusted computing base. *That is, the certificate can be checked and trusted even if the model checker is itself not trusted*.

Name	*LOC*	*CVC*	*Vampyre*	*SAT*	*Cert*	*Core*	*Improve*
ide.c	7428	80720	x	**100**	**703**	>2000	807
ide.c	7428	82653	x	**100**	**1319**	>2000	827
tlan.c	6523	11145980	x	**517**	**4663**	>200	21559
tlan.c	6523	90155057	x	**572**	**74281**	>200	157614
aha152x.c	10069	247435	x	**210**	**2102**	>1500	1178
aha152x.c	10069	247718	x	**210**	**3968**	>1500	1180
synclink.c	17104	9822	x	**53**	**185**	>500	185
synclink.c	17104	9862	x	**53**	**327**	>500	186
hooks.c	30923	597642	x	**369**	**2004**	>1500	1629
hooks.c	30923	601175	x	**368**	**3102**	>1500	1624
cdaudio.c	17798	248915	156787*	**209**	**2006**	>1000	750
diskperf.c	4824	117172	x	**106**	**955**	>2500	1105
floppy.c	17386	451085	60129*	**318**	**2595**	>3000	189
kbfiltr.c	12131	56682	7619*	**51**	**528**	>2500	149
parclass.c	26623	460973	x	**262**	**2156**	>4500	1759
parport.c	61781	2278120	102967*	**529**	**3568**	>5000	195
SSL-srvr	2483	1287290	19916	**261**	**1055**	>150	76
SSL-clnt	2484	189401	27189	**155**	**740**	>200	175
Micro-C	6272	416930	118162	**262**	**2694**	>5500	451
Micro-C	6272	435450	x	**263**	**7571**	>5500	1656

Table 6-1: Comparison of Proof Certificate Size SAT vs. Conventional Theorem Provers

6.6.3 Separation Logic for Predicate Abstraction

Predicate abstraction and counterexample-guided abstraction refinement have become de rigor for software model checking. Each of these technologies relies fundamentally on some variant of Hoare logic to define the execution semantics of program statements. However, Hoare logic has limited usefulness in the presence of shared data structures and aliasing. In these situations the compositional nature of the logic breaks down. This presents two alternatives to developers of model-checking technology: (1) sidestep the issue by concentrating on verification of properties that are not dependent on data and aliasing; or (2) use preprocessing technology that can detect and possibly eliminate aliasing. The first alternative

limits the effective range of the technology, the second introduces the potential for unsound results—which is at variance with the objective of certifying model checking.

Magill and Nanevski have, under the partial support of this SEI research, investigated the use of Reynolds's separation logic as a basis for predicate abstraction [Reynolds 02]. Separation logic is a substructural logic that can be used to reason about finite resources—such as the properties of memory heaps and pointer aliasing. Magill and Nanevski have developed a proof theory for a subset of separation logic sufficient to reason about non-trivial pointer properties programs. Technical details on this work have been documented in [Magill 05]. If this innovative result can be generalized to the C language (and there are no technical reasons to think otherwise), then certifying model checkers such as those developed by the SEI could be parameterized by logical systems specifically adapted to the properties of interest.

6.7 References

URLs are valid as of the publication date of this document.

[Appel 00] Appel, A. & Felty, A. "A Semantic Model of Types and Machine Instructions for Proof-Carrying Code." Annual Symposium on Principles of Programming Languages. *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Boston, MA, 2000: 243-253.

[Appel 01] Appel, A. "Foundational Proof-Carrying Code." *16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*. June 2001.

[Chaki 04a] Chaki, Sagar; Clarke, Edmund; Groce, Alex; Jha, Somesh; & Veith, Helmut. "Modular Verification of Software Components in C." *IEEE Transactions on Software Engineering (TSE)*, 30(6): 388–402, June 2004.

[Chaki 04b] Chaki, S.; Clarke, E.; Ouaknine, J.; Sharygina, N.; & Sinha, N. "State/Event-Based Software Model Checking," 128-147. Integrated Formal Methods 4th International Conference (IFM 2004) (in *Lecture Notes in Computer Science* [LNCS], volume 2999). Canterbury, Kent, UK, April 4-7, 2004. Berlin, Germany: Springer-Verlag, 2004.

[Chaki 05] Chaki, Sagar. "SAT-Based Software Certification." *In submission*.

[Dachuan 04] Dachuan, Yu & Zhong, Shao. Verification of Safety Properties for Concurrent Assembly Code. *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*. Snowbird, Utah, September 2004, 175-188.

[Henzinger 02] Henzinger, Thomas; Jhala, Ranjit; Majumdar, Rupak; Necula, George; Sutre, Gregoire; & Weimer, Westley. "Temporal Safety Proofs for Systems Code." *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*. Copenhagen, July 2002.

[Magill 05] Magill, S.; Nanevski, A.; Clarke, E.; & Lee, P. "Inferring Invariants in Separation Logic for Imperative List-Processing Programs." *In submission*.

[Moskewicz 01] Moskewicz, Matthew W.; Madigan, Conor F.; Zhao, Ying; Zhang, Lintao; & Malik, Sharad. Chaff: Engineering an Efficient SAT Solver. *Proceedings of the 38th Design Automation Conference, (DAC 2001)*, Las Vegas, NV, June 18-22, 2001. ACM 2001, 530–535.

[Namjoshi 01] Namjoshi, Kedar S. "Certifying Model Checkers." *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '01)*. Lecture Notes In Computer Science, Vol. 2404. London, UK, Springer-Verlag, 2001.

[Necula 96a] Necula, G. & Lee, P. "Safe Kernel Extensions Without Run-Time Checking." Proceedings of the USENIX 2nd Symposium on OS Design and Implementation (OSDI '96), Oct. 28-31, 1996, Seattle, WA, 229-243.

[Necula 97] Necula, G. "Proof-Carrying Code." *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 15-17 January 1997. ACM Press, New York, 1997, 106-119.

[Necula 01a] Necula, G. "A Scalable Architecture for Proof-Carrying Code." Invited paper at the Fifth International Symposium on Functional and Logic Programming, Tokyo, March 2001.

[Necula 01b] Necula, G.C. & Rahul, S.P. "Oracle-Based Checking of Untrusted Software." *SIGPLAN Notices 36*, No. 3 (March 2001): 142-54.

[Peled 01] Peled, Doron & Zuck, Lenore. "From Model Checking to a Temporal Proof." Proceedings of SPIN 2001. *Lecture Notes in Computer Science, Vol. 2057*. Springer, Berlin Heidelberg New York, 2001.

[Schneck 02a] Schneck, R. & Necula, G. "A Gradual Approach to a More Trustworthy, yet Scalable, Proof-Carrying Code." *Proceedings of the 18th International Conference on Automated Deduction (CADE '02)*, Copenhagen, Denmark, July 27-30, 2002. Lecture Notes in Computer Science 2392, Springer, 2002.

[Schneck 02b] Schneck, R. & Necula, G. "Proof-Carrying Code with Untrusted Proof Rules." *Proceedings of the 2nd International Software Security Symposium*. November 2002.

[Reynolds 02] Reynolds, John C. "Separation Logic: A Logic for Shared Mutable Data Structures." *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*. Copenhagen, Denmark, July 2002, 55–74.

[Zhong 02] Zhong, Shao. "A Type System for Certified Binaries." *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, OR, January 16-18, 2002. ACM SIGPLAN Notices 37(1), January 2002, 217-232.

7 Verification of Evolving Software via Component Substitutability Analysis

Sagar Chaki, Edmund Clarke, Natasha Sharygina, Nishant Sinha

7.1 Introduction

Correctness of computer software is critical in today's information society. This is especially true of software that runs on computers embedded in our transportation and communication infrastructure. Examples of serious software errors are easy to find. For instance, in 1997 the propulsion system of the Aegis missile cruiser *USS Yorktown* failed for more than two hours because of a software bug [Slabodkin 98]. The cause turned out to be a division by zero within a database system, which resulted in an exception and crashed all computer consoles and terminal units. The software of the *USS Yorktown* operated on a network of Windows NT machines and was quite complex, consisting of several million of lines of C code.

Another instance is the development of the F/A-22 as part of the Joint Strike Fighter program. The project was delayed multiple times, often because the software was not ready. Pilots often had to reboot computers while in the air [Nellemann 94]. The F/A-22 has about 2.5 million lines of software written in Ada. This number is expected to rise to 6 million lines of C/C++ code on the F-35.

Computer software also plays an important role in other parts of our infrastructure. On August 14, 2003, a blackout affected more than 50 million people in large areas on the U.S. east coast, causing estimated damage between $4 billion and $10 billion [USCPSOTF 04]. While the blackout was triggered by trees hitting local power transmission lines, it was a software bug that made it devastating. A bug in GE Energy's XA/21 power control system allowed the blackout to spread. The software had been in use since 1990, but the bug had never before become apparent. The flaw was discovered by an audit of more than 4 million lines of C/C++ code after the blackout and was identified as a "race condition."

Programs in imperative languages such as C or C++ are executed line-by-line in what is called the *thread of control*. It is tempting to hope that a line-by-line inspection of the code, following this thread of control, will uncover all the flaws in a program. The problem is that complex systems have many software components running in parallel, so that there are many different threads of control that run simultaneously. While one of these threads may currently be executing some statement in its program, another thread with exactly the same program,

may be executing an entirely different line of code. Consequently, in the presence of multiple threads, one has to consider any combination of program lines the threads can execute.

Thread A		Thread B	
1	while(x!=0) skip;	1	while(x!=0) skip;
2	x=1;	2	x=2;
3	3		

Figure 7-1: A Small Program with Two Threads of Control

The *state* of the program is the location of the control in each thread, and the values of the program variables. In order to discover flaws, it is necessary to explore the possible states of the program. To illustrate the large number of states that concurrency can cause, consider the small program in Figure 7-1. It has one variable *x*, which is initialized with zero. It has two threads *A* and *B* of control, and only four lines of code in total. The first line in both threads simply idles until *x* becomes zero. The second line set *x* to 1 or 2, respectively. Despite its tiny size, the program has 10 reachable states. The blowup is due to the different combinations of program locations in the two threads *A* and *B*. Thus, a manual search for errors in large concurrent programs is infeasible.

Model checking [Clarke 81, Clarke 99] is an automated technique for exploring all the states of a system. Introduced in 1981, it has become a standard verification technique in the hardware industry. It has been successfully used to find bugs in circuitry that would have been hard to find by inspection.

Model checking has the potential to produce major enhancements in the reliability and robustness of software as well. The basic idea of software model checking is to explore all the states of the software system systematically. The states are checked for errors. Such an error may be division by zero as in the case of the *USS Yorktown*, a race condition as in the case of GE's XA/21, or a violated assertion. Once such an erroneous state is found, it can be reported to the programmer together with a counterexample, that is, an error trace, that demonstrates the flaw. Counterexamples can be very helpful for understanding the nature of the error and fixing it.

However, the effectiveness of model checking such systems is severely constrained by the state space explosion problem, that is, by the sheer number of states a program can be in. If there are too many states, it becomes impossible to explore all of them, even on a powerful computer.

Much of the research in this area is therefore targeted at reducing the state-space of the model used for verification. One principal method in state space reduction of software systems is abstraction. Abstraction techniques reduce the program state space by generating a smaller set of states in a way that preserves the relevant behaviors of the system. Abstractions are most often performed in an informal, manual manner, and require considerable expertise.

Manual abstraction is error-prone, too. The person performing the abstraction will often capture the intended behavior when abstracting, and not the behavior of the actual code, and thus, could hide a bug. Industrial applications of model checking therefore favor automated ways to compute the abstract model. One such method, called *predicate abstraction* [Graf 97, Colon 98], has proven to be particularly successful when applied to large software programs. We exploited predicate abstraction while developing a solution to a problem of establishing correctness of evolving systems.

The other principal approach in reducing the state-space of the verifiable model is *compositional reasoning*. Compositional reasoning partitions verification into checks of individual modules, while the global correctness of the composed system is established by constructing a proof outline that exploits the modular structure of the system. We used the *assume-guarantee* style of compositional reasoning [Pnueli 85] to support verification of evolved systems.

7.2 Model Checking

In formal verification, a system is modeled mathematically, and its specification (also called a claim in model checking) is described in a formal language. When the behavior in a system model does not violate the behavior specified in a claim, the model satisfies the specification. Model checking [Clarke 82] is a fully automated form of formal verification that uses algorithms that check whether a system satisfies a desired claim through an exhaustive search of all possible executions of the system. The exhaustive nature of model checking renders the typical testing question of adequate coverage unnecessary.

Model checking is a technique for verifying finite-state concurrent systems. One benefit of restricting ourselves to finite-state systems is that verification can be performed automatically. Given sufficient resources, model checking always terminates with a yes or no answer. Moreover, it can be implemented by algorithms that have reasonable efficiency and that can be run on moderate-sized machines.

Although the restriction to finite-state systems may seem to be a major disadvantage, model checking is applicable to several very important classes of systems [Clarke 99]. Hardware controllers are finite state systems, as are many communication protocols. Software, which is not finite-state, may still be verified if variables are assumed to be defined over finite domains. This assumption does not restrict the applicability of model checking because many interesting behaviors of the software systems can be specified with finite-state models. For example, systems with unbounded message queues can be verified by restricting the size of the queues to a small number like two or three.

In classical model checking, systems are modeled mathematically as state transition systems and claims are specified using temporal logic [Pnueli 97, Clarke 86]. Temporal logic is used to define formulas that describe system behavior over time, where the propositions of the logic are behaviors of interest involving state information (current state or values of vari-

ables) or events. Temporal logic formulas combine such propositions with temporal operators to describe interesting patterns of propositions over time, such as

- Whenever *X* is greater than *Y*, *Z* must also be greater than *Y*.
- Some invariant (e.g., mutual exclusion with respect to some resource) always holds once initialization is complete.
- A component can only issue requests during an allowed interval (as bounded by events granting and taking away permission).

Temporal logic model checking is extremely useful in verifying the behavior of systems composed of concurrent processes or interacting nondeterministic sequential tasks. Concurrency errors (as well as errors caused by the nondeterministic execution of actions) are among the most difficult to find by testing because they tend to be nonreproducible.

7.3 The Process of Model Checking

Model checking involves the following steps:

4. The system is modeled using the description language of a model checker, producing a model *M*.
5. The claim to check is defined using the specification language of the model checker, producing a temporal logic formula ϕ.
6. The model checker automatically checks whether $M \models \phi$ satisfies.

The model checker checks all system executions captured by the model and outputs the answer *yes* if the claim holds in the model (*M*) and the answer *no* otherwise. When a claim is not satisfied, most model checkers produce a *counterexample* of system behavior that causes the failure. A counterexample defines an execution trace that violates the claim. Counterexamples are one of the most useful features of model checking, as they allow users to quickly understand why a claim is not satisfied.

7.4 Current Research in Software Model Checking

Model checking is efficient in hardware verification, but applying it to software is complicated by several factors, ranging from the difficulty of modeling computer systems (because of the complexity of programming languages as compared to hardware description languages) to difficulties in specifying meaningful claims for software using the usual temporal logical formalisms of model checking. The most significant limitation, however, is the *state space explosion problem* (which applies to both hardware and software), whereby the complexity of model checking becomes prohibitive.

State space explosion results from the fact that the size of the state transition system is exponential in the number of variables and concurrent units in the system. When the system is composed of several concurrent units, its combined description may lead to an exponential blowup as well. The state space explosion problem is the subject of most model checking research.

The following is the list of the state-space reduction techniques commonly used during verification of software.

- *Compositional reasoning*. Verification is partitioned into checks of individual modules while the global correctness of the composed system is established by constructing a proof outline that exploits the modular structure of the system.
- *Abstraction*. A smaller abstract system is constructed such that the claim holds for the original system if it holds for the abstract system.
- *Counterexample-guided abstraction refinement*. Abstracted systems are refined iteratively using information extracted from counterexamples until an error is found or it is proven that the system satisfies the verification claim.

7.4.1 Compositional Reasoning

Because model checking was created for verifying hardware systems and since most hardware designs have a natural division into modules, the extension of model checking to larger designs was accomplished by taking a *divide-and-conquer* approach. Under the approach, we decompose the verification claim of the system into a set of local claims of the system modules and verify them separately. The compositional approach aims to establish whether for given systems $M1$, $M2$ and a claim T, the composed system satisfies T (written $M1 \parallel M2 \models T$).

A naive compositional approach proceeds by executing the following steps: (1) $M1 \models T$ and (2) $M2 \models T$, and concludes by proofs that $M1 \parallel M2 \models T$. Though this rule is sound in theory, it is often not useful in practice—both $M1$ and $M2$ usually behave like T only in a suitable environment. To solve this problem, the compositional principle can be strengthened to an *assume-guarantee* principle [Abadi 95, Alur 96, Clarke 89, Kurshan 95, McMillan 97]: in order to check $M \models T$, it suffices to check both $M1 \parallel T2 \models T1$ and $M2 \parallel T1 \models T2$. This obligation uses the local specifications $T1$, $T2$ as the constraining environment (also called assumptions) with regard to the behavior of $M2$, $M1$ taken in isolation from $M1$, $M2$ respectively. In general, for a system composed of multiple modules, assume-guarantee reasoning succeeds as long as it can be shown that each system component M_i satisfies a corresponding specification component T_i under a suitable constraining environment.

7.4.2 Abstraction

Abstraction is one of the principal complexity reduction techniques [Ball 01, Clarke 92, Kurshan 94]. Abstraction techniques reduce the state space by mapping the concrete set of actual system states to an abstract set of states that preserve the actual system's behavior. Abstractions are usually performed in an informal, manual manner and require considerable expertise. Predicate abstraction [Graf 97, Colon 98] is one of the most popular and widely applied methods for the systematic abstraction of systems. It maps concrete data types to abstract data types through predicates over the concrete data. However, the computational cost of the predicate abstraction procedure may be too high, making generation of a full set of predicates for a large system infeasible.

In practice, the number of computed predicates is bounded, and model checking is guaranteed to deliver sound results within this bound. The bound limit is increased once errors (if any) are found within the bound and fixed. Under this approach, software systems are rendered finite by restricting variables to finite domains. As mentioned earlier, bounded model checking does not seriously restrict the applicability of model checking because many interesting behaviors of software systems can be specified using bounded finite-state models.

The abstract program is created using *existential abstraction* [Clarke 92]. This method defines the transition relation of the abstract program so it is guaranteed to be a *conservative* over-approximation of the original program, with respect to the set of given predicates. The use of a conservative abstraction, as opposed to an *exact* abstraction, produces considerable reductions in the state space. The drawback of the conservative abstraction is that when model checking of the abstract program fails, it may produce a counterexample that does not correspond to a concrete counterexample. Such a counterexample is usually called *spurious*. When a spurious counterexample is encountered, *refinement* is performed by adjusting the set of predicates in a way that eliminates it.

7.4.3 Counterexample-Guided Abstraction Refinement (CEGAR)

Though conservative abstraction procedures—which ensure that if a claim holds for the abstract system, it also holds for the original system—are typically used, any form of abstraction may introduce behaviors not found in the concrete system. Counterexamples from model checking the abstract system are often used to detect unrealistic behaviors and refine the system. Repeatedly refining the abstractions, however, may introduce additional behaviors that result in state space explosion during the model checking phase. These drawbacks—coupled with the potential effectiveness of abstraction methods—motivate research into targeted abstractions (i.e., control abstraction, loop abstraction, and so forth), which can result in more accurate abstract systems.

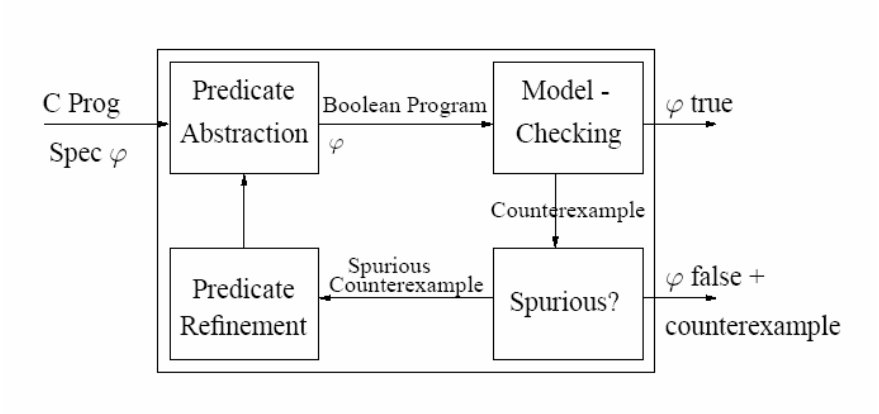

Figure 7-2: The CEGAR Framework

The abstraction refinement process has been automated by the CEGAR paradigm [Kurshan 94, Ball 00, Clarke 00, Das 01]. The CEGAR framework is shown in Figure 7-2: one starts with a coarse abstraction (for example, an abstraction of a C program). If an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the *abstract-verify-refine* paradigm and depend on the abstraction and refinement techniques used. The steps are described below in the context of predicate abstraction.

1. **program abstraction**. Given a set of predicates, a finite state model is extracted from the code of a software system, and the abstract transition system is constructed.
2. **verification**. A model checking algorithm is run to check whether the model created by applying predicate abstraction satisfies the desired behavioral claim φ. If the claim holds, the model checker reports success (φ is true), and the CEGAR loop terminates. Otherwise, the model checker extracts a counterexample, and the computation proceeds to the next step.
3. **counterexample validation**. The counterexample is examined to determine whether it is spurious. This examination is done by simulating the (concrete) program using the abstract counterexample as a guide, to find out if the counterexample represents an actual program behavior. If this is the case, the bug is reported (φ is false), and the CEGAR loop terminates. Otherwise, the CEGAR loop proceeds to the next step.
4. **predicate refinement**. The set of predicates is changed to eliminate the detected spurious counterexample and possibly other spurious behaviors introduced by predicate abstraction. Given the updated set of predicates, the CEGAR loop proceeds to Step 1.

The efficiency of this process is dependent on the efficiency of the program abstraction and predicate refinement procedures. While program abstraction focuses on constructing the transition relation of the abstract program, the focus of predicate refinement is to define efficient techniques for choosing the set of predicates in a way that eliminates spurious counterexam-

ples. In both areas of research, low computational cost is a key factor since it enables the application of model checking to the verification of realistic programs.

This project presented techniques that use efficient abstraction and abstraction refinement techniques of the CEGAR loop by employing techniques implemented in the Copper model checker [Chaki 05b]. In this project we presented a solution to the model checking problem that arises during verification of evolving systems. This solution was originally published by Chaki [Chaki 05a]. We refer the reader to earlier work by Chaki [Chaki 04b] for details regarding the Copper abstraction and refinement procedures.

7.5 Verification of Evolving Software

Successfully transitioning model checking technology has proven to be a challenging task. While the benefits of successful model checking are clear, there are several barriers to successful transition. Principally, model checking has serious scalability problems and the techniques are difficult for software engineers to use.

A major short-coming in most model checking research is the failure to consider how to make model checking use routine throughout various stages of software development. Software inevitably evolves as designs take shape, requirements change, and bugs are discovered and fixed. Model checking is useful at each such point, but the current state of model checking requires that software verification of the entire system be performed anew each time. The amount of time and effort required to verify an entire system can be considerable and repeating the exercise after each change, no matter how small, would likely discourage use.

This report presents results of considering ways to reduce the effort of subsequent verifications. In particular, by exploiting the results of previous verification efforts and focusing only on the portions of the system that have changed (components), model checking can be incorporated into development processes in a much less intrusive or cumbersome manner.

We present techniques that, while not affecting the initial model checking effort, reduce by orders of magnitude the effort to keep analysis results up-to-date with evolving system design. The presented techniques are decision procedures that determine if all system correctness properties previously established by model checking remain valid for the new version of the system.

The key idea is to determine automatically if these properties hold for the new system without repeating each of the individual verification checks. We will present a verification method [Chaki 05a] that focuses on system components that have changed during the evolution of software and determines if all behaviors of the original system are preserved in the new version of the system. Moreover, whenever it is found that behaviors are not preserved, whenever possible our technique will automatically provide feedback to developers showing how to improve the components.

Upgrade#(Prop.)	# Mem. Queries	T_{orig} (msec)	T_{ug} (msec)
$ipc_1(P_1)$	279	2260	13
$ipc_1(P_2)$	308	1694	14
$ipc_2(P_1)$	358	3286	17
$ipc_2(P_2)$	232	805	10
$ipc_3(P_1)$	363	3624	17
$ipc_3(P_2)$	258	1649	14
$ipc_4(P_1)$	355	1102	24

Figure 7-3: Comparison of Times Required for Original Verification (T_{orig}) and Verification on Upgrade (T_{ug}) by DynamicCheck

7.6 Implementation and Experimental Evaluation

We implemented the dynamic component substitutability check procedures in the COPPER model checker. The tool includes a front end for parsing and constructing control-flow graphs from C programs. Further, it is capable of (1) model checking properties on programs based on automated may-abstraction (existential abstraction), and (2) it allows compositional verification by employing learning-based automated assume-guarantee reasoning. We reused the above features of COPPER in the implementation of the substitutability check. The tool interface was modified so that a collection of components and corresponding upgrades could be specified. We extended the learning-based automated assume-guarantee to obtain its dynamic version, as required in the compatibility check. This involved keeping multiple learner instances across calls to the verification engine and implementing algorithms to validate multiple previous observation tables in an efficient way during learning. We have also implemented the under-approximation generation algorithms for carrying out containment check on small program examples. This involved procedures for implementing must-abstractions from C code using predicates obtained from C components. The automated refinement procedures are still under implementation and would enable containment check of larger benchmarks.

We validated the component substitutability framework while verifying upgrades of a benchmark provided to us by our industrial partner, ABB Inc. The benchmarks consist of seven components that together implement an interprocess communication (IPC) protocol. The combined state-space is over 10^6.

We used a set of properties describing functionality of the verified portion of the IPC protocol. We used upgrades of the *write-queue* (ipc_1) and the *ipc-queue* (ipc_2 and ipc_3) components. The upgrades had both missing and extra behaviors compared to their original versions. We verified two properties (P_1 and P_2) before and after the upgrades. We also verified the properties on a simultaneous upgrade (ipc_4) of both the components. P_1 specifies that a process may write data into the *ipc-queue* only after it obtains a lock for the corresponding critical section. P_2 specifies an order in which data may be written into the *ipc-queue*. Table 1 shows the comparison between the time required for initial verification of the IPC system with the time taken by DynamicCheck for verification of upgrades. In Figure 7-3 *#Mem.*

Queries denotes the total number of membership queries made during verification of the original assembly.

We observed that the previously generated assumptions in all the cases were sufficient to prove the properties on the upgraded system also. Hence, the compatibility check succeeded in a *small fraction of time* (T_{ug}) as compared to the time for compositional verification (T_{orig}) of the original system.

7.7 Related Work

Related projects often impose the restriction that every behavior of the new component must also be a behavior of the old component. In such a case the new component is said to refine the old component. For instance, de Alfaro et al. [de Alfaro 01, Chakrabarti 02] define a notion of interface automaton for modeling component interfaces and show compatibility between components via refinement and consistency between interfaces. However, automated techniques for constructing interface automata from component implementations are not presented. In contrast, our approach automatically extracts conservative DLA models (which are similar to finite state interface automata) from component implementations. Moreover, we do not require refinement among the old components and their new versions.

Ernst et al. [McCamant 04] suggest a technique for checking compatibility of multi-component upgrades. They derive consistency criteria by focusing on input/output component behavior only and abstract away the temporal information. Even though they state that their abstractions are unsound in general, they report success in detecting important errors. In contrast, our abstractions preserve temporal information about component behavior and are always sound. They also use a refinement-based notion on the generated consistency criteria for showing compatibility.

The application of learning is extremely useful from a pragmatic point of view since it is amenable to complete automation, and is gaining rapid popularity [Groce 02] in formal verification. The use of learning for automated assume-guarantee reasoning was proposed originally by Cobleigh et al. [Cobleigh 03]. The use of learning along with predicate abstraction has also been applied in the context of interface synthesis [Alur 05] and various types of assume-guarantee proof rules for automated software verification [Chaki 04a].

This work is related to our earlier project [Chaki 04c] that solves the component substitutability problem in the context of verifying *individual* component upgrades. A major improvement of the current work is that it is aimed at verifying the component substitutability in the presence of *simultaneous upgrades of multiple components*. Another distinction of this work is that it provides an innovative *dynamic* assume-guarantee reasoning framework for the compatibility check. The dynamic nature of the compatibility check allows reusing previously computed assumptions to prove or disprove the global properties of the updated system.

Additionally, this paper gives a new solution to the containment check problem that Chaki presented [Chaki 04c]. In our earlier work, the containment step is solved using learning techniques for regular sets and handles finite-state systems only. In contrast, the new approach is extended to handle infinite-state C programs. Moreover, this paper defines a new technique based on simultaneous use of over and under approximations obtained via existential and universal abstractions.

7.8 Conclusion

The SEI independent research and development project on the verification of evolving software via component substitutability analysis addressed a critical and vital problem of component substitutability analysis and provided a solution that consists of two phases: *containment* and *compatibility* checks. The compatibility check performs compositional reasoning with the help of a *dynamic* regular language inference algorithm and a model checker. Our experiments confirm that the dynamic approach is more effective than complete re-validation of the system after an upgrade. The containment check detects behaviors that were present in each component before but not after the upgrade. These behaviors are used to construct useful feedback to the developers. We observed that the order of components used to discharge the assume-guarantee rules has a significant impact on the algorithm runtimes and hence needs investigation.

The component substitutability analysis has been implemented in a Copper tool [Chaki 05b] that can be invoked within the ComFoRT framework. The verification framework was validated on an industrial benchmark provided by our industrial partner, ABB Ltd., and demonstrated encouraging results.

7.9 References

URLs are valid as of the publication date of this document.

[Abadi 95] Abadi, M. & Lamport, L. “Conjoining Specifications.” *ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 17*, No. 3 (May 1995).

[Alur 96] Alur, R. & Henzinger, T. “Reactive Modules, 207-218. *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. New Brunswick, NJ, 27-30 July 1996. IEEE Computer Society Press, 1996.

[Alur 05] Alur, R.; Cerny, P.; Gupta, G.; Madhusudan, P.; Nam, W.; & Srivastava, A. “Synthesis of Interface Specifications for Java Classes.” *Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Long Beach,

CA, Jan. 12-14, 2005.

[Angluin 87] Angluin, D. "Learning Regular Sets from Queries and Counterexamples." *Information and Computation, Volume 75*, 2 (1987): 87–106.

[Ball 00] Ball, T. & Rajamani, S. "Boolean Programs: A Model and Process for Software Analysis (2000-14)." Microsoft Research, February 2000.

[Ball 01] Ball, T.; Majumdar, R.; Millstein, T.; & Rajamani, S. "Automatic Predicate Abstraction of C Programs." *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. Snowbird, UT, 2001.

[Chaki] Chaki, S. "Learning Doubly Labeled Automata Using Queries and Counterexamples." Unpublished manuscript. Available at http://www.sei.cmu.edu/staff/chaki/publications/learn-se-trace.pdf.

[Chaki 04a] Chaki, S.; Clarke, E.; Giannakopoulou, D.; & Pasareanu, C. S. *Abstraction and Assume-Guarantee Reasoning for Automated Software Verification (05.02)*. Research Institute for Advanced Computer Science (RIACS), 2004.

[Chaki 04b] Chaki, S.; Clarke, E.; Groce, A.; Ouaknine, J.; Strichman, O.; & Yorav, K. "Efficient Verification of Sequential and Concurrent C Programs." *Formal Methods in System Design Volume 25*, Issue 2-3 (September-November 2004): 129-166.

[Chaki 04c] Chaki, S.; Sharygina, N.; & Sinha, N. "Verification of Evolving Software." *Proceedings of the Third International Workshop on Specification and Verification of Component-Based Systems*. 2004.

[Chaki 05a] Chaki, S.; Clarke, E.; Sharygina, N.; & Sinha, N. "Dynamic Component Substitutability Analysis." *Proceedings of the Formal Methods 2005 Conference*, 2005.

[Chaki 05b] Chaki, S.; Ivers, J.; Sharygina, N.; & Wallnau, K. "The ComFoRT Reasoning Framework." *Lecture Notes in Computer Science, Volume 3576*. Springer, 2005

[Chakrabarti 02] Chakrabarti, A.; de Alfaro, L.; Henzinger, T. A.; Jurdzinski, M.; & Mang, F. Y. C. "Interface Compatibility Checking for Software Modules," 428–441. *Proceedings of the 14th International Conference on Computer Aided Verification*. 2002. *Lecture Notes in Computer Science, Volume 2404*,.

[Clarke 81] Clarke, E. M. & Emerson, E. A. "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic." *Logic of Programs: Workshop*. Springer-Verlag, 1981.

[Clarke 82] Clarke, E. M. & Emerson, E. A. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic." *Logic of Programs Workshop*, 1982.

[Clarke 86] Clarke, E. M.; Emerson, E. A.; & Sistla, A. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications." *ACM Transactions on Programming Languages and Systems*, 1986.

[Clarke 89] E. Clarke, D. L. & McMillan, K. "Compositional Model Checking," 353-362. *Proceedings of the Fourth Symposium on Logic in Computer Science*. IEEE Computing Society Press, 1989.

[Clarke 92] Clarke, E.; Grumberg, O.; & Long, D. "Model Checking and Abstraction." *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1992.

[Clarke 99] Clarke, E.; Grumberg, O.; & Peled, D. *Model Checking*. MIT Press, December 1999.

[Clarke 00] Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; & Veith, H. "Counterexample-Guided Abstraction Refinement." *Journal of the ACM (JACM), Volume 50*, Issue 5 (September 2003): 752-794.

[Cobleigh 03] Cobleigh, J. M.; Giannakopoulou, D.; & Pasareanu, C. S. "Learning Assumptions for Compositional Verification." *Tools and Algorithms for Construction and Analysis of Systems* (in Lecture Notes in Computer Science [LNCS], volume 2619. Springer-Verlag, 2003.

[Colon 98] Colon, M. & Uribe, T. "Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures." *Computer Aided Verification*. 1998: 293–304.

[Das 01] Das, S. & Dill, D. "Successive Approximation of Abstract Transition Relations." *Proceedings of the 16th Annual IEEE Symposium*

on Logic in Computer Science (LICS). IEEE, 2001.

[de Alfaro 01] de Alfaro, L. & Henzinger, T. A. "Interface Automata." *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. 2001.

[DoD 05] U.S. Department of Defense. F-35 Joint Strike Fighter Program. http://www.jsf.mil/ (2005).

[Giannakopoulou 02] Giannakopoulou, D.; Pasareanu, C. S.; & Barringer, H. "Assumption Generation for Software Component Verification." *Proceedings of the 17th IEEE International Conference on Automated Software Engineering 2002 (ASE 2002)*. Edinburgh, Scotland, Sept. 23-27, 2002. IEEE, 2002.

[Graf 97] Graf, S. & Saidi, H. "Construction of Abstract State Graphs with PVS," 72–83. *Proceedings of the Ninth International Conference on Computer Aided Verification (CAV'97)* (in Lecture Notes in Computer Science [LNCS] volume 1254, Grumberg, O., ed. Springer-Verlag, 1997.

[Groce 02] Groce, A.; Peled, D.; & Yannakakis, M. "Adaptive Model Checking," 357–370. *Tools and Algorithms for Construction and Analysis of Systems*. Springer-Verlag, 2002.

[Kurshan 94] Kurshan, R. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton, NJ:Princeton University Press, 1994.

[Kurshan 95] Kurshan, R. *Computer-Aided Verification of Coordinating Processes*. Princeton, NJ: Princeton University Press, 1995.

[McCamant 04] McCamant, S. & Ernst, M. D. "Early Identification of Incompatibilities in Multi-Component Upgrades." *Proceedings of the Object-Oriented Programming, 18th European Conference (ECOOP 2004)* (in Lecture Notes in Computer Science [LNCS], volume 3086. Oslo, Norway, June 14-18, 2004. Springer, 2004.

[McMillan 97] McMillan, K. "A Compositional Rule for Hardware Design Refinement." *Proceedings 9th International Conference on Computer Aided Verification (CAV'97)* (in Lecture Notes in Computer Science [LNCS] volume 1254. Haifa, Israel, 1997. Springer-Verlag, 1997.

[Nellemann 94] Nellemann, D. "Air Force F-22 embedded computers", September 1994. http://archive.gao.gov/t2pbat2/152615.pdf.

[Pnueli 85] Pnueli, A. “In Transition from Global to Modular Temporal Reasoning About Programs.” *Logics and Models of Concurrent Systems*. New York, NY: Springer-Verlag New York, Inc., 1985: 123-144.

[Pnueli 97] Pnueli, A. “The Temporal Logic of Programs,” 46-57. *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*. Providence, RI, October 31-November 2, 1997. New York, NY: Institute of Electrical and Electronics Engineers, Inc. (IEEE), 1997.

[Rivest 93] Rivest, R. L. & Schapire, R. E. “Inference of Finite Automata Using Homing Sequences.” *Information and Computation, Volume 103*, 2 (1993): 299–347.

[Roscoe 97] Roscoe, A. W. *The Theory and Practice of Concurrency*. Prentice-Hall International, 1997.

[Slabodkin 98] Slabodkin, Gregory. “Navy: Calibration flaw crashed Yorktown LAN.” *Government Computer News*. Nov. 9, 1998; Vol. 17 No. 30. Available at http://www.gcn.com/17_30/news/33914-1.html.

[USCPSOTF 04] U.S.-Canada Power System Outage Task Force. *Final Report on the August 14, 2003 Blackout in the United States and Canada*. April 2004. https://reports.energy.gov/.

8 Emerging Technologies and Technology Trends

Angel Jordan

8.1 Introduction

This is the second report from the Software Engineering Institute (SEI) on emerging technologies and technology trends in the fields of software engineering and systems engineering. The first report was published in October 2004 as part of the SEI technical report *Results of SEI Independent Research and Development Projects and Report on Emerging Technologies and Technology Trends* (CMU/SEI-2004-TR-018).

As mentioned in the first report, technology scouting has always been an implicit activity of the SEI and is embedded in the SEI's mission of technology transition. Because of the institute's small size relative to other research institutions, the SEI applies the most leverage to its active initiatives, but it also watches for other emerging technologies in the U.S. and internationally.

In this report as in the first one, we present information about technologies that are pushing the frontiers of the SEI's current programs and initiatives, as well as technologies that transcend them. The SEI Independent Research and Development (IR&D) program, described earlier in this document, is an example of explicit technology scouting at the SEI. The past activities of the SEI New Frontiers Group, including information collection and dissemination, are further examples.

As in the first report, we also mention the activities of the International Process Research Consortium (IPRC). The purpose of the IPRC is to develop a community of practice that regularly collaborates to examine and codify future process research opportunities and directions. IPRC members come from all over the world, bringing expertise in process research and a vision for the trends, challenges, and needs for software-intensive organizations over the next 5-10 years.

In the first report, we provided descriptions of new or emerging technologies.[4] These descriptions included the technologies' purpose and origin. Where possible, we indicated the technologies' level of maturity and provided information about related trends. A bibliography for

[4] More detailed white papers, written by SEI technical staff members, are available for some of these technologies. To obtain copies, contact SEI Customer Relations at 412-268-5800.

the technology descriptions was provided at the end of the report. The following technologies were described:

- open grid services architecture
- integrated security services for dynamic coalition management
- model-driven architecture
- service-oriented architecture
- automated lexical and syntactical analysis in requirements engineering
- Q methodology
- emergent algorithms for interoperability
- aspect-oriented software development
- generative programming
- software assurance
- recent advances in intrusion detection systems
- advances in software engineering processes

The technical staff members of the SEI continue to follow closely all of these technologies.

In the section on Advances in Software Engineering Processes we reported in the first report notable advances and trends as follows, which we repeat here for emphasis.

8.1.1 Reducing Software Defects to Improve Security

It is recognized that defective software is not secure, a position advocated by the SEI and a few other organizations (e.g., PRAXIS and Cigital), and accepted by the Department of Homeland Security (DHS) Software Process Subgroup of the Task Force on Security. This position is supported by the fact that the leading cause of software vulnerabilities is common defects in software design and implementation (i.e., bugs). Also, tools for developing secure software, although needed, are not sufficient and address only a small part of the problem. Formal methods, better processes, and training for software professionals will have more impact and are critically needed. The DHS subgroup made recommendations in this context. The reader is referred to the first report for a detailed description of these recommendations.

Other trends described in the first report, which have renewed relevance, are:

- use of tabular expressions
- stratified systems theory
- model-based process improvement, including

  - deploying Six Sigma, the SEI Team Software Process (TSP) and Personal Software Process (PSP), and Agile with the Capability Maturity Model Integration (CMMI) methodology
  - increasing use of the Project Management Body of Knowledge (PMBOK) to improve management competencies. (Project Management as contained in the PMBOK is in many respects complementary to the methods articulated in Software Improvement as developed by the SEI and others.)
- increasing efforts to harmonize various systems and software standards. This is another recognizable trend. Indicators include
  - efforts by IEEE to harmonize its standards with ISO and CMMI
  - efforts by ISO to harmonize its standards related to quality as well as integrating systems and software
- wider use of appraisal methods, including more quantification of process improvement

8.1.2 Organization of this Report

After these introductory paragraphs, this report presents a self-contained section with its own introduction, titled "Technology Scouting of Work at Carnegie Mellon University and Other Institutions Worldwide Relevant to SEI." The main report then follows with a section titled "Technology Scouting in Systems and Software Engineering," is also a self-contained report with its own introduction, conclusions, and references.

The main report then follows with a section titled "International Workshops on Software Process." Here we report on workshops that are organized by an international community of software engineers, both from academe and industry worldwide, and that provide forums for the latest advances in software process improvement. The list of participants and program organizers reads almost like a "who's who in software engineering," even though the programs are centered mainly on software development processes.

The main report then contains a section titled "Agile Software Development." Many members of the technical staff of the SEI are familiar with the Agile Software Development community, through the SEI has no presence in that community. Developments there deserve to be followed because they have an impact on the practice of software engineering and reflect significant advances in software technology.

Finally, this report presents a section describing another forum on software engineering titled "International Conferences in Software Engineering." These conferences provide the vehicle for presenting the most recent advances in software technology.

The report employs HTML links that the reader can follow to additional details as desired. These links serve as references in most cases.

8.2 Technology Scouting of Work at Carnegie Mellon University and Other Institutions Worldwide Relevant to SEI

This section describes work at the Carnegie Mellon University campus (mainly in the School of Computer Science [SCS]) which is relevant to some of the programs in the Software Engineering Institute. It dwells briefly on some areas within SEI where there is synergy with the work carried out on campus. In both cases the work described here is believed to be at the frontier of software technology development. It cites work in other institutions that are doing related work by leading researchers and, where the information is available, indicates the state of the art in the field and/or the state of maturity.

In preparation for this section, the author conducted interviews with a number of faculty members in the School of Computer Science and at the SEI. The interviews at the SEI were conducted to corroborate the author's knowledge of some of the fields acquired either before or during his tenure as Acting SEI Director.

The section contains a number of opinions or judgments by certain individuals, which sometimes are subjective. This is the case, for instance, when statements such as *prominent people, leaders in the field, working at the frontier*, etc. are made.

The section often borrows freely from Web sites of institutions, and from personal Web pages of researchers in universities, and thus the writing style reflects that of the people who are being quoted. In lieu of references, links to the Web sites or Web pages are given. This allows the readers of the section to visit the sites at the same time that the reading is done. This is particularly easy to do if this document is read online in its HTML version or onscreen in its PDF version, where the hyperlinks have been maintained.

The section starts with a subsection titled "Advances in Software Architecture." Under this heading, an innovative project called ArchJava, directed by a faculty member in SCS, is described, followed by a brief enumeration of a number of architecture description languages (ADLs), defined to describe, model, check, and implement software architectures. An enumeration of work related to ArchJava follows. The section continues with a subsection on work on software architecture being done at SCS, at the SEI, and at other universities. It then follows with a subsection on aspect-oriented programming and aspect-oriented software development. The topic that is being watched by SEI members is briefly introduced and is followed by a description of an innovative research project led by a faculty member in SCS, which is attracting the attention of the SEI. In the next subsection, the report describes work on autonomic application software, a field that is attracting the attention of the software community internationally, and is relevant to the SEI. A section on verification of autonomous systems follows. This work done at SCS is at the frontier of software research and is the source of a fruitful collaboration between SCS and the SEI. The section proceeds with a subsection on proof-carrying code, a pioneering work in SCS, which also is the source of collaboration between SCS and the SEI. Then the section follows with a brief description of an

innovative project, the Fox Project, also at SCS, that may have an impact on the SEI. The section culminates with a description of a study being conducted at the National Academies (with participation of members of Carnegie Mellon), which is to have an impact in the development of software technology, and should be observed and followed by the SEI.

8.2.1 Advances in Software Architecture

Software architecture is a field of software engineering and computer science where Carnegie Mellon, in the School of Computer Science and at the SEI, is at the leading edge in software technology. At the SEI, the work in software architecture is principally conducted in the Product Line Systems Program, albeit other programs or initiatives also are engaged in software architecture. In SCS the work is done in the Institute of Software Research International (ISRI) and in the Computer Science Department, which are both in the School of Computer Science. In this subsection we briefly describe work at Carnegie Mellon that is advancing the state of the art in software architecture, briefly summarize work at the SEI, and mention other institutions that are also working at the frontier of the field.

8.2.1.1 ArchJava

This work is pursued at Carnegie Mellon University by Jonathan Aldrich (http://www.cs.cmu.edu/~aldrich/) and his students. The work originated from a foundation in programming languages but is centered in software architecture.

In the words of Jonathan Aldrich:

> *Software architecture describes the structure of a system, enabling more effective design, program understanding, and formal analysis. However, existing approaches decouple implementation code from architecture, allowing inconsistencies, causing confusion, violating architectural properties, and inhibiting software evolution. ArchJava is an extension to Java that seamlessly unifies software architecture with implementation, using a type system to ensure that the implementation conforms to architectural constraints.*
> *(http://archjava.fluid.cs.cmu.edu/index.html).*

The site above includes a downloadable compiler for ArchJava as well as publications describing the language, a case study, and the theory behind ArchJava. The work originated with Jonathan Aldrich's doctoral thesis (Jonathan Aldrich, Using Types to Enforce Architectural Structure. University of Washington Ph.D. Dissertation, August 2003, available at http://archjava.fluid.cs.cmu.edu/papers/aldrich-dissertation.pdf.)

Other people whose work relates to Aldrich's include researchers at the University of Washington led by David Notkin (http://www.cs.washington.edu/homes/notkin/), a Carnegie Mellon alumnus, who are experts in programming languages.

Related work to ArchJava is incorporated in a number of architecture description languages (ADLs), defined to describe, model, check, and implement software architectures. Many of these languages support sophisticated analysis and reasoning or support architecture-centric development. Some recent ADLs include:

- **Wright**, which provides a formal basis for architectural description in software design. This language can be used to provide a precise, abstract meaning to an architectural specification and to analyze both the architecture of individual software systems and of families of systems (http://www.cs.cmu.edu/afs/cs/project/able/www/wright/).
- **UniCon**, an architectural description language whose focus is on supporting the variety of architectural parts and styles found in the real world and on constructing systems from their architecture descriptions (http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/)
- **Acme**, a simple, generic software ADL that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools (http://www.cs.cmu.edu/~acme/)
- **Aesop**, which provides a toolkit for rapidly building software architecture design environments, specialized for domain-specific architectural styles. It consists of an open tool integration framework that supports cooperation between Aesop itself and other tools. It also provides, among other features, a repository for storing, retrieving, and reusing architectural design elements (http://www.cs.cmu.edu/afs/cs/project/able/www/aesop).

Other ADLs are C2 (http://www.isr.uci.edu/architecture/c2.html), CUSADL (http://www.isr.uci.edu/architecture/adl/SADL.html), Darwin (http://www-dse.doc.ic.ac.uk /Software/), MetaH (http://www.htc.honeywell.com/metah/prodinfo.html), Rapide (http://pavg.stanford.edu/rapide/), SADL (http://www.csl.sri.com/programs/dsa /sadl-main.html), and xArch (http://www.isr.uci.edu/architecture/xarch/).

8.2.1.2 Work in software architecture at the Carnegie Mellon School of Computer Science

Frontier work on software architecture is also conducted in other projects at the School of Computer Science at Carnegie Mellon incorporated in an umbrella project called the ABLE Project (http://www.cs.cmu.edu/afs/cs/project/able/www/able.html). ABLE stands for architecture-based languages and environment.

The ABLE project is concerned with exploring the formal basis for software architecture, developing the concept of architectural style, and building tools that practicing software architects might find useful. The tool development effort has focused on the Aesop system (see above). The formal work revolves around the Wright language (see above).

Another project under the umbrella of ABLE is titled "Reasoning about Implicit Invocation Systems" (http://www.cs.cmu.edu/afs/cs/project/able/www/implinvoc/ii.html). It provides a formal basis for reasoning about systems designed using the implicit invocation architectural

style. It replaces current ad hoc reasoning approaches used by practitioners who use the implicit invocation style with a collection of sound ideas that allow better informal reasoning about such systems.

Some papers written by members of the ABLE group are provided through the following links:

- Software Architecture in General (http://www.cs.cmu.edu/afs/cs/project/able/www/able/general)
- Formal Aspects of Software Architecture and Architectural Style (http://www.cs.cmu.edu/afs/cs/project/able/www/able/#formal-section)
- Aesop Software Architecture Design Environments (http://www.cs.cmu.edu/afs/cs/project/able/www/able/#aesop)
- The Acme Architecture Description and Interchange Language (http://www.cs.cmu.edu/afs/cs/project/able/www/able/#acme)
- Working Papers (http://www.cs.cmu.edu/afs/cs/project/able/www/able/#working)
- The Armani Software Architecture Design Environment and ADL (http://www.cs.cmu.edu/afs/cs/project/able/www/able/#armani)
- Pervasive Computing (http://www.cs.cmu.edu/afs/cs/project/able/www/able/#pervasive)
- Others (http://www.cs.cmu.edu/afs/cs/project/able/www/able/#others)

The Defense Advanced Research Projects Agency (DARPA, http://www.darpa.mil/) is the major sponsor of this project at Carnegie Mellon.

The locus of the software architecture projects at the Software Engineering Institute is the Product Line Systems Program. The Web site of this program (http://www.sei.cmu.edu/architecture/sw_architecture.html) nicely articulates what software architecture is all about:

> *Software architecture forms the backbone for building successful software-intensive systems. An architecture largely permits or precludes a system's quality attributes such as performance or reliability. Architecture represents a capitalized investment, an abstract reusable model that can be transferred from one system to the next. Architecture represents a common vehicle for communication among a system's stakeholders, and is the arena in which conflicting goals and requirements are mediated. The right architecture is the linchpin for software project success. The wrong one is a recipe for disaster.*

The SEI's Product Line Systems Program is a leading source of knowledge and expertise in software architecture. It has contributed courses in a comprehensive curriculum, which can be used toward certificate programs; offers an extensive range of architecture-related prod-

ucts and services; and has published a collection of highly acclaimed books on software architecture.

Ongoing work concentrates in the following areas: architecture design, architecture documentation, architecture evaluation, architecture life-cycle evaluation, architecture reconstruction, and reasoning about software quality attributes. The program emphasizes that software architecture is one of the key reusable assets that form the basis of a software product line and is codifying best architecture practices in the context of software product line practice, while helping organizations apply them.

The architecture work is carried out under the auspices of three technical initiatives: Software Architecture Technology (SAT, http://www.sei.cmu.edu/architecture/sat_init.html), Product Line Practice (PLP, http://www.sei.cmu.edu/productlines/plp_init.html), and Predictable Assembly from Certifiable Components (PACC, http://www.sei.cmu.edu/pacc/pacc_init.html).

An example of frontier work on software architecture can be found in a project funded by the SEI IR&D program titled "Architecture-Based Self-Adapting Systems," described earlier in this document. It has been funded for continuation beyond a feasibility study stage this year. It is an excellent example of the collaboration of the SEI with faculty and students in SCS. The investigators are Rick Kazman and David Garlan, assisted by Hong Yan (ISRI PhD student) and Bradley Schmerl (ISRI system scientist).

The goal of the research is to automatically determine the run-time architecture of a system without reverse engineering and to use this to create a reflection model that allows a system to reason about its own behavior and adapt to a changing environment and needs.

At this stage the work is beyond extending a prototype system built by Yan to extract run-time information from a system previously developed by Kazman. It is expected that government agencies working with the researchers will have their system analyzed. Publications, internal to SEI and external are forthcoming.

8.2.1.3 Architecture work in other universities

At the University of California at Irvine, Professor Nikil Dutt's group is doing research that lies at the intersection of compilers, architectures, and computer-aided design, with a specific focus on the exploration, evaluation, and design of domain-specific embedded systems. This group has developed a novel architectural description language that facilitates rapid exploration of programmable embedded systems, as well as automatic generation of software toolkits supporting embedded systems development (including optimizing compilers and simulators). See http://www.ics.uci.edu/%7Edutt/**.**

At the University of Southern California, Professor Barry Boehm and his group are pursuing research that focuses on value-based software engineering, including a method for integrating a software system's process models, product models, property models, and success models called model-based (system) architecting and software engineering (MBASE).

Boehm's contributions to the field include the constructive cost model (COCOMO), the spiral model of the software process, the Theory W (win-win) approach to software management and requirements determination, the foundations for the areas of software risk management and software quality factor analysis, and two advanced software engineering environments: the TRW Software Productivity System and the Quantum Leap Environment**.** See http://sunset.usc.edu/people/barry.html.

At the University of Texas at Austin, Dewayne Perry and his group are pursuing research on software engineering and architecture with goals of establishing principles about, and improving practices for, building and evolving large-scale software and process systems. In his theoretical work, Perry looks for fundamental mechanisms, such as the role of feedback and control in evolution processes and the role of architecture in system evolution. In his empirical work, he primarily uses the results to prune his theoretical work, but also creates empirical methods when needed to support that work. An effect of his many interactions with developments is the transfer of practical insights about their products and processes. See http://www.ece.utexas.edu/faculty/directory/details.php?id=77.

At Imperial College in London, under Jeff Magee in the Department of Computing, work in software architecture deals with architectural description languages, dynamic architectures, and self-organizing architectures. The Distributed Software Engineering section conducts research on the software development process and software support environments, particularly for real-time, embedded, parallel and distributed systems. See http://www.doc.ic.ac.uk/~jnm/.

At the University of Washington, David Notkin's educational and research interests are in software engineering, with a particular focus in software evolution—understanding why software is so hard and expensive to change, and in turn reducing those difficulties and costs. See http://www.cs.washington.edu/homes/notkin/.

At the **University of St. Andrews at Edinburgh**, a strong group of researchers is conducting work on software architecture at the frontiers of the field. Their Web site at http://www.dcs.st-and.ac.uk/research/architecture/ contains a very descriptive, graphic way of defining software architecture:

> *Software architecture is concerned with how to design software components and make them work together. For example, the mechanisms by which enterprises implement their IT strategy by gluing together software components are attracting the attention of system modelers, tool makers and computational joiners (who do the gluing). New methods such as open distributed object systems, process modeling and novel network architectures are being used heavily in industry to address these problems.*
>
> *As the use of a computer system grows it gathers information and accretes users who have an expectation of the manner in which the system may be used. Thus information flows not only between computers but also between computers and*

humans, and humans and humans. To change such a system requires that consideration of the impact of the changes on the users be understood as well as the technological mechanisms for evolution.

8.2.2 Aspect-Oriented Programming (AOP) and Aspect-Oriented Software Development (AOSD)

Aspect-oriented programming (AOP) attempts to provide a clean separation of concerns, enabling programmers to reason about and evolve programs more effectively. Many language constructs have been proposed to enable better separation of concerns, and a number seem to be promising ways to improve the way software development is done. However, a number of open problems remain, including understanding the formal foundations of aspects, supporting aspect encapsulation, understanding automated and human reasoning about aspect-oriented programs, and studying the practical consequences of the technology.

An excellent articulation of aspect-oriented programming is found on the Web site of the Software Engineering Research Group at the University of British Columbia (http://www.cs.ubc.ca/labs/spl/):

A key goal of software design is the separation or modularization of concerns. Many concerns—including error-checking strategies, design patterns, synchronization policies, resource sharing, distribution and performance optimization—crosscut the program structure. When standard procedural or object-oriented programming (OOP) languages are used, it is hard to separate crosscutting concerns. Aspect-oriented programming (AOP) has been developed to support modularization of these concerns. AOP languages provide mechanisms that crosscut program structure in well-defined ways. These mechanisms make it possible to cleanly capture the structure of crosscutting concerns, making both the code and the design easier to understand and develop.

Aspect-oriented software development (AOSD) is a promising emerging technology. AOSD addresses problems experienced with object-oriented development, but has much greater applicability across software development in general. This is not a mature technology but its large-scale adoption by IBM promises to greatly accelerate its maturation. This is not the first time that AOSD has been scouted by the SEI. A series of white papers by the Product Line Systems Program under Linda Northrop, which had the objective to do technology scouting relevant to software product lines, last year included one white paper on this subject. An excerpt from that paper was included in the SEI report *Results of SEI Independent Research and Development Projects and Report on Emerging Technologies and Technology Trends* (CMU/SEI-2004-TR-018), which was co-authored by this writer. The report is available at http://www.sei.cmu.edu/publications/documents/04.reports/04tr018.html.

The following are excerpts from the original white paper:

> *The commitment of industry including HP, IBM, and BEA may well hasten the maturation of AOSD. Their use of the technology will more quickly uncover the gaps of knowledge that only appear when solving industrial-strength problems. The focus on e-commerce and Web servers will also hasten the discovery of relevant design patterns.*
>
> *Because of the strong connections between aspect-oriented software development with software architecture and software product lines, the SEI has been carefully monitoring developments in this field as it has evolved over the past few years. The Product Lines System Program is already involved with the AOSD community. It needs to elicit their support to make needed software tools a reality. Further investigation of the connections between AOSD and software architecture and software product lines is required.*

8.2.2.1 Modular aspect-oriented programming

At Carnegie Mellon, work on aspect-oriented programming by Jonathan Aldrich is incorporated in a project designated *modular aspect-oriented programming* whose goal is to move aspect-oriented programming toward the mainstream of both language design and engineering practice. The work focuses on a number of research questions: how to formally model aspect-oriented programming constructs; how to design a module system; how to design new language features; what are the practical benefits and drawbacks of proposed aspect-oriented language features; how to compare different language designs when applied to similar problems; and how to effectively analyze aspect-oriented programs.

So far, a formal model of aspect-oriented programming named TinyAspect has been built and used to study open modules, a new module system that preserves the extensibility of aspects along with a strong encapsulation property, and can be used to benefit the reasoning benefits of aspect-oriented programming tools.

The work above is on a research stage, but a number of publications by Aldrich and coworkers are in print or have been submitted for publication:

- Jonathan Aldrich. Open Modules: Modular Reasoning about Advice. Submitted for publication. An earlier version appeared in Foundations of Aspect Languages, March 2004. The full proofs for the theorems in the paper are published as Carnegie Mellon Technical Report CMU-ISRI-04-141, December 2004 (which supersedes an earlier technical report, CMU-ISRI-04-108).
- Jonathan Aldrich. "Open Modules: Reconciling Extensibility and Information Hiding." Proceedings of the AOSD 2004 Workshop on Software Engineering Properties of Languages for Aspect Technologies, March 2004. Available at http://www.cs.cmu.edu /~aldrich/papers/splat04.pdf. A file describing the raw, detailed results of our micro-

experiment with the SpaceWar program is available at http://www.cs.cmu.edu/~aldrich /aosd/spacewar-study-details.txt.

- Neel Krishnaswami and Jonathan Aldrich. Statically-Scoped Exceptions: A Typed Foundation for Aspect-Oriented Error Handling. Available at http://www.cs.cmu.edu/~aldrich /papers/static-exceptions.pdf. This article has been submitted for publication.

Prominent people in this field, in addition to those at Carnegie Mellon, include Adrian Colyer at IBM; faculty and students at the University British Columbia; and a group of faculty and students at Northeastern University in the Demeter project (http://www.ccs.neu.edu /research/demeter/).

Adrian Colyer is an IBM Senior Technical Staff Member and the leader of the AspectJ and AspectJ Development Tools Projects on Eclipse.org. He divides his time between working on AO technologies, and helping groups throughout IBM to adopt and apply them. AspectJ is a seamless aspect-oriented extension to the Java programming language. It is Java platform compatible and easy to learn and use. See http://eclipse.org/aspectj/.

At Northeastern University, the Center for Software Science, led by professors Karl Lieberherr and David Lorenz, are conducting research whose objective is to create software that is easy to maintain and evolve using adaptive programming and aspect-oriented programming. See http://www.ccs.neu.edu/research/demeter/index.html.

At the University of British Columbia, a strong group of researchers led by professors Gregor Kiczales, Gail Murphy, and Kris De Volder in the Software Practices Laboratory are advancing the state of the art in aspect-oriented programming. Their work has been cited above.

Another project mentioned previously is the Rapide project at Stanford University (http://pavg.stanford.edu/rapide/).

The Rapide Language effort focuses on developing a new technology for building large-scale, distributed multi-language systems. This technology is based upon a new generation of computer languages, called Executable Architecture Definition Languages (EADLs), and an innovative toolset supporting the use of EADLs in evolutionary development and rigorous analysis of large-scale systems. Rapide is designed to support component-based development of large, multi-language systems by utilizing architecture definitions as the development framework. This effort is led by David C. Luckham (http://pavg.stanford.edu/people/dcl/).

8.2.3 Autonomic Application Software

Autonomic computing aims to reduce the complexity of managing software systems. To be autonomic, a system must configure and reconfigure itself, continually optimize itself, recover from malfunction, or protect itself, while keeping its complexity hidden from the user.

Understanding software engineering issues is critical for the proliferation of autonomic applications.

This field has attracted a number of researchers in the international software community. A workshop in this topical area took place as part of the 27th International Conference on Software Engineering (ICSE, http://www.cs.wustl.edu/icse05/Home/index.shtml). The topical area was titled DEAS 2005: Design and Evolution of Autonomic Application Software (http://www.deas2005.cs.uvic.ca). The organizers, including two members of Carnegie Mellon (one from SCS and the other from the SEI) were David Garlan, SCS; Marin Litoiu, IBM Canada; Hausi A. Muller, University of Victoria, Canada; John Mylopoulos, University of Toronto, Canada; Dennis B. Smith, SEI; and Kenny Wong, University of Alberta, Canada.

The goal of this workshop was to bring together researchers and practitioners who investigate concepts, methods, techniques, and tools to design and evolve autonomic software. While there are several workshops that deal with autonomic computing systems, there are few workshops that focus on software engineering issues—that is, how do we design, build, and evolve such software systems so that they can meet given, and evolving, requirements for particular classes of users and/or applications. Most existing systems cannot be redesigned and redeveloped from scratch to incorporate autonomic capabilities. Rather, self-management capabilities have to be added gradually and incrementally, one aspect at a time. With the proliferation of autonomic applications, users will impose ever-greater demands with respect to functional and non-functional requirements for autonomicity.

Topics of interest in this area include, but are not limited to, architectural styles, attribute-based architectural styles, and architecture patterns for autonomic elements and systems, designing high-variability software, designing self-managed systems, evolving autonomic software, injecting autonomicity into legacy systems, integration mechanisms, methods for evaluating complex tradeoffs, adoption of autonomic systems, or assessing the user experience in self-managed systems.

8.2.4 Verification of Autonomous Systems

This portion of this report is based on an extensive interview with Ed Clarke in the Carnegie Mellon University Department of Computer Science and a follow-up analysis by the writer of the report. When appropriate, pieces of the report are extracted from the Web sites of Clarke and members of his team.

Clarke and his team are working toward developing tools and techniques to support formal verification of autonomous systems. This work is highly synergistic with, and has an influence on, related work at the SEI. The overall project is conducted under the umbrella of "automatic verification of computer hardware and software."

8.2.4.1 Automatic verification of computer hardware and software

The rationale for this work is the recognition that logical errors in sequential circuit designs and communication protocols constitute important problems for system designers. They can delay considerably the deployment of new products to the market or cause the failure of some critical device that is already in use. The research group under Clarke has developed a verification method called temporal logic model checking for this class of systems. In this approach, specifications are expressed in a propositional temporal logic, while circuits and protocols are modeled as state-transition systems. An efficient search procedure is used to determine automatically if a specification is satisfied by some transition system. The technique has been used to find subtle errors in a number of cases.

The size of the state-transition systems that can be verified by model checking techniques has recently increased dramatically. By representing transition relations implicitly using binary decision diagrams (BDDs), cases have been checked that would have required 10^{20} states with the original algorithm. Various refinements of the BDD-based techniques have pushed the state count up to 10^{100}. By combining model checking with various abstraction techniques, it is possible to handle even larger systems. For example, the technique has been used to verify the cache coherence protocol in the IEEE Futurebus+ standard. Several errors were found that had been previously undetected.

For additional information see the Carnegie Mellon University Model Checking home page (http://www.cs.cmu.edu/%7Emodelcheck/).

The overall project led by Clarke is a collaborative effort between SCS and the Automated Software Engineering Group at NASA Ames Research Center. See Automated Software Engineering Group (http://ase.arc.nasa.gov/).

At NASA Ames, the goal of "robust software engineering" is to increase by orders of magnitude both the quality and the productivity of software engineering. The cross-cutting research done by the Robust Software Engineering group of Code TI (http://ic.arc.nasa.gov/) at NASA Ames (http://www.arc.nasa.gov) draws upon several disciplines, including: artificial intelligence—particularly automated reasoning and knowledge representation; formal methods; programming language theory; mathematical logic; and advanced compiler methods. The focus is on strategic research—that is, research that is directed to the 5–15-year time horizon, aiming to make large impacts rather than incremental advances. The research is done in the context of pacing NASA applications, as a means of both providing feedback to the group and as a means for the group to make contributions to NASA's goals as progress is made. The group currently has space-related projects in space science code generation, and in software verification for deep-space missions. For civilian aviation, the group is engaged in research on next-generation auto-coding technology and high-assurance software design. They are also developing innovative educational technology, and have some sample lessons for students and teachers based on work done so far.

This work is advancing the state of the art and although it is moving at a rapid pace, it has not reached yet a state of practice. The technology will reach a state of maturity several years before it becomes ready for transition.

(At the time of this writing, the writer of this report is still looking at people in other institutions doing related work.)

It is worth mentioning that the work of Clarke was the source of a fruitful collaboration between SCS and SEI: the IR&D project titled "Verification of Evolving Software via Component Substitutability Analysis," which is described earlier in this report.

8.2.5 Proof-Carrying Code

This work is led by Peter Lee in SCS. Proof-carrying code (PCC) is a technique by which a code consumer (e.g., a host) can verify that code provided by an untrusted code producer adheres to a predefined set of safety rules. These rules, also referred to as the safety policy, are chosen by the code consumer in such a way that they are sufficient guarantees for safe behavior of programs. There are many potential applications of PCC. For example, for mobile code the code consumer would be an Internet host (e.g., a Web browser) and the code producer a server that sends applets. In operating systems, one can have the kernel act as the host, with untrusted applications acting as code producers that download and execute code in the kernel's address space.

The key idea behind proof-carrying code is that the code producer is required to create a formal safety proof that attests to the fact that the code respects the defined safety policy. Then, the code consumer is able to use a simple and fast proof validator to check, with certainty that the proof is valid and hence the foreign code is safe to execute. See http://www-2.cs.cmu.edu/~petel/papers/pcc/pcc.html.

This work, which is considered frontier research, has recently attracted the attention of the SEI, and led to collaboration between Kurt Wallnau from the SEI and Peter Lee. The collaboration was funded in the IR&D project titled "Proof-Carrying Code," described earlier in this report.

8.2.6 The ConCert Project

8.2.6.1 Certified Code for Grid Computing

The ConCert Project, also led by Peter Lee, investigates the theoretical and engineering basis for the trustless dissemination of software in an untrusted environment. To make this possible the project investigates machine-checkable certificates of compliance with security, integrity, and privacy requirements. Such checkable certificates allow participants to verify the intrinsic properties of disseminated software, rather than extrinsic properties such as the software's point of origin.

To obtain checkable certificates the project develops certifying compilers that equip their object code with formal representations of proofs of properties of the code. Specifically, the project investigates the use of proof-carrying code, typed intermediate languages, and typed assembly languages for this purpose. In each case certificate verification is reduced to type-checking in a suitable type system.

To demonstrate the utility of trustless software dissemination, the project develops an infrastructure for building applications that exploit the computational resources of a network of computers. The infrastructure consists of a "steward" running on host computers that accepts and verifies certified binaries before installing and executing them, and certifying compilers that generate certified binaries for distribution on the network. See http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/concert/www/.

8.2.7 The Fox Project

This project, whose principal investigators are professors Robert Harper, Peter Lee, and Frank Pfenning of SCS, is also deemed to be at the frontier of advanced research and promises to have an impact in software development technology. It is funded by DARPA.

The objective of the Fox Project is the development of language support for building safe, highly composable, and reliable systems. It seeks to accomplish this by exploiting and advancing the state of the art in programming language technology, including fundamental design principles, compiler technologies, and the mathematical underpinning of programming languages and logics. Results are demonstrated through language implementations and applications in systems software, such as embedded systems or active networks, emphasizing those that must simultaneously be highly customizable, safe, and efficient.

The current emphasis is on applications for program composition in embedded systems. See http://www-2.cs.cmu.edu/~fox/.

8.2.8 Building Certifiably Dependable Software Systems

Under the Current Projects System of the National Academies, a study project that merits close follow-up by the SEI for its relevance and future impact is titled "Sufficient Evidence? Building Certifiably Dependable Software Systems." Peter Lee is a member of the committee conducting this study.

This project will convene a mixed group of experts to assess current practices for developing and evaluating mission-critical software, with an emphasis on dependability objectives. The committee will address system certification, examining a few different application domains (e.g., medical devices and aviation systems) and their approaches to software evaluation and assurance. This should provide some understanding of the common ground and disparities that exist. The discussion will engage members of the fundamental research community, who have been scarce in this arena. It will consider approaches to systematically assessing systems' user interfaces. It will examine potential benefits and costs of improvements in evalua-

tion of dependability as performance dimensions. It will evaluate the extent to which current tools and techniques aid in ensuring and evaluating dependability in software and investigate technology that might support changes in the development and certification process. It will also use the information amassed to develop a research agenda for dependable software system development and certification, factoring in earlier high-confidence software and systems research planning. It will also investigate ideas for improving the certification processes for dependability-critical software systems.

The work of the committee is being conducted in two phases. Phase I consisted of a workshop and summary report, completed early in 2004. Phase II follows on from the framing provided by Phase I and should result in a final report to be issued at the end of the project.

This project is funded by the National Science Foundation. See http://www4.nas.edu/webcr.nsf/5c50571a75df494485256a95007a091e/a4362b7f9a6cc0c685256da4004cc0f4?OpenDocument&Highlight=0,dependable.

8.3 Technology Scouting in Systems and Software Engineering

8.3.1 Introduction

In this section of the report, we look at researchers and institutions that publish in journals related to systems and software engineering. It is well known that the two areas, systems engineering and software engineering, are interrelated and sponsors of the SEI are increasingly interested in aspects of software engineering and technology that should be more interlinked with systems (as broadly defined by the Department of Defense).

Members of the SEI over the years have reported with some pride the high ranking of the institute in the field of systems and software engineering. Other institutions highly ranked also tend to publicize their high rankings. The SEI consistently ranks as number one. It is interesting and prudent to analyze the value of this ranking, which comes from a publication titled *An Assessment of* Systems *and Software Engineering Scholars and Institutions* by Robert L. Glass and T.Y. Chen. (Glass and Chen, 2005). This publication purports to name the top scholars and institutions in the field of systems and software engineering. It presents the findings of a five-year study of the top scholars and institutions in the *systems* and software engineering field, as measured by the quantity of papers published in the journals of the field. The top scholar is Khaled El Emam of the Canadian National Research Council, and the top institution is Carnegie Mellon University and its Software Engineering Institute. The publication is part of an ongoing study, conducted annually, that identifies the top 15 scholars and institutions in the most recent five-year period. It attempts to answer the following questions: Who are the most published scholars in the field of systems and software engineering (SSE)? And which are the most published institutions? As stated in the publication:

> *The paper is the 11th in an annual series whose goal is to answer those questions. The first such paper was (Glass, 1994); subsequently such studies have been published each year, in a fall issue of the* Journal of Systems and Software *(when the journal was published 12 times per year, the study findings were published in the October issue; now that it is published 15 times per year, they are published in the 12th or 13th issue). This is the seventh year in which the study has included five years worth of data (in the previous years, 1–4 years were covered). In future years, the study will continue to cover the most recent five year period.*

The publication also states: "This paper reports on the top scholars and institutions for the five-year period 1999–2003. The methodology of the study and its limitations are discussed in the article, which further adds: "It is important to note two things at the outset, however: (1) The study findings are based on frequency of publication in the leading journals in the SSE *field*. (2) The study focuses on the *field* of SSE, and not, for example, on computer science or information systems."

With these caveats and limitations the findings are of value to those of us who have been following the field of software engineering for some time and who also pay attention to the field of systems engineering, particularly because of the importance of the interrelations between systems engineering and software engineering. It is also interesting to note that a number of the top scholars and institutions highly ranked would not be on the radar screen of those of us who periodically scout the field of software engineering.

In this report we look at the leading scholars and institutions that are highly ranked in Glass and Chen's study. We look at the list of the names and provide the available links to Web sites of authors and/or institutions. In some cases, we give the personal Web page of the researcher and also of his/her department or school. In other cases, we provide access to biographies. After quoting the conclusions drawn by Glass and Chen in their own study, we draw our own conclusions.

8.3.1.1 Top scholars in the field of systems and software engineering

Khaled El Emam, Canadian National Research Council. He is also associated with the Cutter Consortium, which has been for some time a proponent of Agile Software Development (see Agile Software Development & Project Management Practice, http://www.cutter.com /project/practice.html). El Emam's stated interests are software quality and software measurement. (Agile software development is described in more detail later in this report.)

Barbara Kitchenham, Computer Science, Keele University, UK (http://www.cs.keele.ac.uk /main.php?page=home&menu=home&content=home). Kitchenham is a well-known researcher whose interests are in software metrics, project management, quality management, technology evaluation, and evidence-based software engineering.

Hyoung-Joo Kim, Seoul National University, Korea (http://web.cse.snu.ac.kr/english/index.asp and http://oopsla.snu.ac.kr/). Kim is a well-known researcher whose interests are in XML, semantic web, and object-oriented systems.

Robert L. Glass, Computing Trends (http://www.developerdotstar.com/mag/bios/robert_glass.html). Glass is an old timer in the field of software engineering. He is the author of the magazine *developer.* (The Independent Magazine for Software Development Professionals)*. He is an author and consultant on software quality issues who has written more than 20 books on the topic. He owns his own company, Computing Trends, and writes columns on Software Engineering for *ACM Communications Magazine* and *IEEE Software*. He is a co-author of the main reference of this section. His areas of expertise are software problems and solutions, software practice, software as a discipline, and project failure.

Lionel C. Briand, Department of Systems and Computer Engineering, Carleton University, Canada (http://www.sce.carleton.ca/faculty/briand/index.html and http://www.carleton.ca/). Briand's interests are in software testing, empirical software engineering, and object-oriented analysis and design.

Brian Henderson-Sellers, Faculty of Information Technology, University of Technology, Sydney, Australia (http://www.uts.edu.au/). Henderson-Sellers' interests are in object-oriented methodologies, metamodeling, and modeling languages.

Richard Lai, Department of Computer Science & Computer Engineering, La Trobe University, Melbourne, Australia (http://www.latrobe.edu.au/cs/). Lai's interests are in Web services, communication protocol engineering, component based software engineering, software metrics, and testing.

Kassem Saleh, American University, Sharjah, United Arab Emirates (http://www.lyee-project.soft.iwate-pu.ac.jp/en/unit/uae/Kassem_CV-AUS-oct-2002.pdf). Saleh's interests are in distributed systems and software mobility.

Mary Jean Harrold, College of Computing, Georgia Institute of Technology (http://www.cc.gatech.edu/~harrold/). Harrold's interests are in scalable program analysis-based software engineering, regression testing, analysis and testing of object-oriented software, software visualization, and remote monitoring of deployed software.

Claes Wohlin, Software Engineering Research Lab. Blekinge Institute of Technology, Sweden (http://www.ipd.bth.se/cwo/Claes.html and http://www.bth.se/tek/serl/). Wohlin is interested in empirical methods, software metrics, software quality, and systematic improvement.

Myoung Ho Kim, Korea Advanced Institute of Science and Technology, Korea (http://www.kaist.edu/). Kim's interests are database systems and distributed information processing.

T.Y. Chen, Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia (http://www.ict.swin.edu.au/, http://www.it.swin.edu.au/staff/tchen,

and http://www.ict.swin.edu.au/. Chen is interested in software testing, software quality, and software maintenance.

Xudong He, School of Computer Science, Florida International University (http://www.cs.fiu.edu/faculty/hex/ and http://www.cs.fiu.edu/home.php). He's interests are formal methods, software architecture, and software testing.

Per Runeson, Software Engineering Research Group, Lund University, Sweden (http://serg.telecom.lth.se/). Runeson is interested in empirical software engineering, verification and validation, and software quality management.

James A. Whittaker, Center for Software Engineering Research, Florida Institute of Technology (http://www.cs.fit.edu/wds/faculty/whittaker/whittaker.html). Whittaker is interested in computer security, penetration testing, software testing, and software engineering.

Hai Zhuge, Chinese Academy of Sciences (http://www.ict.ac.cn/en/3-33.htm and http://grid.hust.edu.cn/gcc2004/chair_comm.htm). Zhuge is interested in Internet-based software engineering, software process model, knowledge-based software engineering, and team software development.

8.3.1.2 Journals included in the survey

The journals included in the survey are:

- *Information and Software Technology* (IST), Elsevier Science
- *Journal of* Systems *and Software* (JSS), Elsevier Science
- *Software Practice and Experience* (SPE), John Wiley & Sons, UK
- *Software* (SW), IEEE
- *Transactions on Software Engineering and Methodologies* (TOSEM), ACM
- *Transactions on Software Engineering* (TSE), IEEE

8.3.1.3 Leading institutions

The leading 15 institutions in the field, and the journals where their researchers publish, are shown in the following table.

Rank	Institution	Journals
1	Carnegie Mellon/SEI	All
2	Korea Advanced Institute of Science and Technology	All but TOSEM, TSE, SW
3	National Chiao Tung University	All but TOSEM, SW
4	Fraunhofer IESE	All but TOSEM
5	Bell Labs/Lucent	All
6	Seoul National University, Korea	All but TOSEM, SW
7	City University, Hong Kong	All but TOSEM
8	Iowa State University	All but TOSEM, SW
9	Microsoft	All but TOSEM, SW

10	National University of Singapore	All but TOSEM, SW
11	Georgia Institute of Technology	All but SW
12	Lund University, Sweden	All but TOSEM
13	National Cheng Kung University	All but TOSEM, SPE, SW
14	Osaka University	All but SW
15	Aristotle University of Thessaloniki, Greece	All but TOSEM, SW

This table is reproduced from the main reference in this report, but without institutional scores.

Links for the 15 institutions are given below.

- Carnegie Mellon/SEI: http://www.cmu.edu/ and http://www.sei.cmu.edu/
- Korea Advanced Institute of Science and Technology: http://www.kaist.edu/
- National Chiao Tung University: http://www.nctu.edu.tw/english/
- Fraunhofer IESE: http://www.iese.fhg.de/
- Bell Labs/Lucent: http://www.lucent.com/
- Seoul National University, Korea: http://www.snu.ac.kr/engsnu/
- City University, Hong Kong: http://www.cityu.edu.hk/
- Iowa State University: http://www.iastate.edu/
- Microsoft: http://www.microsoft.com/
- National University of Singapore: http://www.nus.edu.sg/
- Georgia Institute of Technology: http://www.gatech.edu/
- Lund University, Sweden: http://www.lu.se/o.o.i.s/450
- National Cheng Kung University: http://www.ncku.edu.tw/english/
- Osaka University: http://www.osaka-u.ac.jp/eng/
- Aristotle University of Thessaloniki, Greece: http://www.auth.gr/index.en.php3

As pointed out in the cited reference, the study is specific to the field of systems and software engineering. For an analysis of the characteristics of research in this field, see Glass et al., (2002). There are similar studies for the related fields of computer science (CS) and information systems (IS). Similar analyses of the research in these fields may be found in Ramesh et al. (2004) for CS and Vessey et al. (2002) for IS. For comparisons of the different studies and results see again the main reference of this report. This reference describes the study methodology, journals, counting schemes, and limitations of these types of studies.

8.3.1.4 Conclusions from the study of Glass and Chen

The study is one in an ongoing series whose goal is to identify the top scholars and institutions in the field of SSE. Similar studies in related fields (CS and IS) convince us that such a study is meaningful and worthwhile. By now, at the end of 11 years of conducting the study, we believe we can identify with some confidence those top scholars and institutions:

Top Scholars:

1. Khaled El Emam of the Canadian National Research Council
2. Barbara Kitchenham of Keele University
3. Hyoung-Joo Kim of Seoul National University, Korea
4. Robert L. Glass, Computing Trends
5. Lionel Briand of Carleton University, Canada

Top Institutions:

1. Carnegie Mellon University and the Software Engineering Institute
2. Korea Advanced Institute of Science and Technology
3. National Chiao-Tung University of Taiwan
4. Fraunhofer Institute for Experimental Software Engineering
5. Bell Labs, Lucent

Regarding the relationship of the field with its collegial fields of CS and IS, we find

- a few similarities with CS in the list of top institutions, but still enough differences to be able to say that SSE is a different field from CS, and
- enough differences with the field of IS to say that they are clearly quite different fields.

(A study of the curriculum and research differences between the fields may be found in Glass (1992, curriculum), and Glass et al. (2004, research).)

8.3.1.5 Further analysis of the study and some observations

When reading the study of Glass and Chen an objective observer would find at first a few surprises. Prominent computer scientists and prominent institutions with high reputations in computer science and computer engineering do not appear in the lists of top scholars or institutions. The reason is quite simple. The researchers of these institutions, although they publish in the journals of the study, do not publish with high frequency in these journals. They tend to publish in archival journals not related to software engineering and systems engineering. They publish in the *Transactions on Software Engineering and Methodologies* of ACM and the *Transactions on Software Engineering* of IEEE, but with much less frequency.

More important is the observation made by Glass and Chen in their own study. They observe, "A few similarities with CS in the list of top institutions exist, but still enough differences to be able to say that SSE is a different field from CS." This is also our own observation. From the beginning of the field of software engineering going back to its foundations, pure computer scientists, even those making fundamental contributions to programming languages, have not regarded software engineering with the same esteem accorded to other branches of computer science. These scientists/engineers hold similar views to a large extent in those aspects of software engineering related to systems engineering.

Those software and systems engineers contributing in such important aspects of the field as process development, quality, testing and measurements, are underrepresented in the higher ranks of the professional associations. This is corroborated by the smaller numbers of Fellows of the IEEE and of ACM compared with those of more mainstream branches of computer science/engineering. This point is even more evident when one looks at memberships in the National Academies. Only a handful of prominent engineers who can be regarded as software engineers are in the National Academy of Engineering. This situation will probably change in due course—indeed it is already changing as funding agencies and, in particular, organizations and entities of DoD, accrue to the field of SSE the prominence it deserves. The distinction goes beyond basic or fundamental versus applied when ascribing these attributes to computer science and software engineering, respectively, because there are many contributions in software engineering and systems engineering that are basic and fundamental in nature.

Another observation worth pointing out is that there are institutions overseas and some institutions in the U.S. that are making notable contributions in systems and software engineering, and are thus worth scouting, but they normally would not be on the radar screen of observers in this country, Looking at those institutions highly ranked in the study of Glass and Chen, and excluding those that would be mentioned as such by a casual observer, it is worth singling out the following institutions:

- Korea Advanced Institute of Science and Technology
- National Chiao Tung University
- Fraunhofer IESE
- Seoul National University, Korea
- City University, Hong Kong
- Iowa State University
- National University of Singapore
- Lund University, Sweden
- National Cheng Kung University
- Osaka University

- Aristotle University of Thessaloniki, Greece

A sequel to this report will concentrate on these institutions and others in Glass and Chen's study as well as those researchers mentioned in the study.

8.3.1.6 References

Glass, Robert L. & Chen, T.Y. "An Assessment of Systems and Software Engineering Scholars and Institutions (1999-2003)." *Journal of Systems and Software 76* (2005): 91-97.

Glass, R.L. "A Comparative Analysis of the Topic Areas of Computer Science, Software Engineering and Information Systems. *Journal of Systems and Software*. November 1992.

Glass, R.L. "An Assessment of Systems and Software Engineering, Scholars and Institu tions." *Journal of Systems and Software*. October 1994.

Glass, R.L.; Ramesh, V.; & Vessey, I. "Research in Software Engineering: An Analysis of the Literature. *Information and Software Technology*. June 1, 2002.

Glass, R.L.; Vessey, I.; & Ramesh, V. "An Analysis of Research in Computing Disciplines." *Communications of the ACM*. June 2004.

Ramesh, V.; Glass, R.L.; & Vessey, I. "Research in Computer Science: An Empirical Study." *Journal of Systems and Software*. February 2004.

Vessey, I.; Ramesh, V.; & Glass, L. "Research in Information Systems: An Empirical Study of Diversity in the Discipline and its Journals." *Journal of Management Information Systems*. Fall 2002.

8.3.2 2005 Software Process Workshop

Observations of international conferences or workshops are a meaningful way of doing scouting on what is going on in the fields of software engineering. This is particularly the case when these conferences or workshops are not attended by a number of active members of the Software Engineering Institute, or take place in remote places not openly accessible to members of SEI.

One such workshop took place in the People's Republic of China, May 25–27, 2005, with the theme "Unifying the Software Process Spectrum" (http://www.cnsqa.com/cnsqa/jsp/html/spw/index.jsp). The lone attendee (as a keynote speaker) from the SEI was Watts Humphrey. Representatives from a number of institutions well known to members of the SEI attended the workshop, either as invited speakers, as presenters of papers in the program, or as members of the program committee. Many of these people are known to the SEI, but others are not. Yet it is worthwhile to track their work, particularly when this work or the authors are not ordinarily on the SEI radar screen.

The report on this workshop starts with the enumeration of the sponsors of the workshop. It follows with the articulation of the need for more research on software process and the quest of those attempting to create a rigorous, orderly discipline of software process engineering, both of which the author of this report finds interesting and eloquent. It follows with a list of participants in the workshop and the invited speakers (with titles of their presentations). This list almost reads as a "who's who in software engineering." The report follows with the program, with authors and the titles of their presentations. The program is divided into different aspects of software process, namely process content, process tools and metrics, process management, process representation and analysis, and experience reports, and culminates with a panel on "Directions in Software Process Research: Where Are We Now, What Should We Do Next," chaired by Leon Osterweil. Through the reports, links are given that may be useful when the names or the institutions are not familiar to the reader.

This workshop attracted our attention due to the caliber of the keynote speakers, the program, and the participation of a number of speakers from the Peoples Republic of China, who often are not on the radar screen of observers from the Western world. The workshop was sponsored by The Institute of Software, Chinese Academy of Science (http://www.iscas.ac.cn/english/index_english.htm and http://www.cnsqa.com/cnsqa/jsp/html/spw/sponsors.jsp).

Cooperating in the sponsorship was the ISCAS Lab for Internet Software Technologies (http://www.cnsqa.com/cnsqa/ShowMainAction.do).

Participating in the sponsorship of the workshop was the well known USC Center of Software Engineering under the direction of Barry Boehm. Members of Boehm's group also made presentations. (http://sunset.usc.edu/cse/index.html).

Expanding on the theme of the workshop, the call for papers and the narrative of the program define in a nice way the nature of the workshop, very much in tune with the articulation of those who recognize the need for more research on software process and those attempting to create a rigorous, orderly discipline of software process engineering:

> *The expanding role of software and information systems in the world has focused increasing attention upon the need for assurances that software systems can be developed at acceptable speed and cost, on a predictable schedule, and in such a way that resulting systems are of acceptably high quality and can be evolved surely and rapidly as usage contexts change. This sharpened focus is creating new challenges and opportunities for software process technology. The increasing pace of software system change requires more lightweight and adaptive processes, while the increasing mission-criticality of software systems requires more process predictability and control, as well as more explicit attention to business or mission values. Emergent application requirements create a need for ambiguity-tolerance. Systems of systems and global development create needs for scalability and multi-collaborator, multi-culture concurrent coordination. COTS products provide powerful capabilities, but their vendor-determined evo-*

lution places significant constraints on software definition, development, and evolution processes.

The recognition of these needs has spawned a considerable amount of software process research across a broad spectrum. Much of the research has addressed the overall characteristics and needs of software processes, focusing on such issues as process architectures, process behavioral characteristics, and how processes fit with higher level organizational systems and characteristics. We refer to these investigations as macroprocess research. Simultaneously there has also been considerable research directed towards the precise, complete, detailed and unambiguous definition of software processes, focusing on such issues as detection of process flaws, and facilitation of the human-machine synergies inherent in software processes. We refer to these investigations as microprocess research. A major goal of this workshop was to suggest ways in which to integrate these two complementary lines of research to create a rigorous, orderly discipline of software process engineering. This integration could suggest, for example, how high level process behaviors might be predicted, and modified, through lower level analyses and optimizations. It could also explore how best to integrate objective microprocesses based on explicit knowledge with more subjective collaboration processes based on tacit knowledge.

This workshop was intended to provide a forum for assessing current and emerging software process capabilities with respect to the challenges, and for obtaining insights into the software process research directions needed to address the challenges and make progress toward overriding goals. It included initial presentations by leading international software process researchers and users, presentations of contributed papers on process challenge areas and solution approaches, tool demonstrations, and a closing panel on software process research directions."

Although the lists of participants in the workshop are not all the prominent people doing research in software process, they almost read as a "who's who in software process":

- **Leon J. Osterweil**, Department of Computer Science, University of Massachusetts
- **Prof. Dr. H. Dieter Rombach**, Head of the Research Group for Software Engineering (AGSE, http://wwwagse.informatik.uni-kl.de/) and of the Fraunhofer Institute for Experimental Software Engineering (IESE, http://www.iese.fhg.de/)
- **Mary Lou Soffa**, Department Chair and Professor of Computer Science
- **Frances Paulisch**, Siemens AG in Munich, Germany. She is responsible for the "Siemens Software Initiative."

- **S.C. Cheung**, Associate Professor of Computer Science, Associate Director of Cyberspace Center, Hong Kong University of Science and Technology (http://www.ust.hk/en/index.html)
- **Jo(anne) M. Atlee**, Director of Software Engineering, Associate Professor School of Computer Science, University of Waterloo (http://www.cs.uwaterloo.ca/)
- **T.Y. Chen**, Professor of Software Engineering, School of Information Technology, Swinburne University of Technology (http://www.swin.edu.au/ict/)
- **Anthony Finkelstein**, Professor of Software Systems Engineering, Head of Department of Computer Science, University College London (http://www.cs.ucl.ac.uk/staff/A.Finkelstein/)
- **Lori A. Clarke**, Department of Computer Science, University of Massachusetts
- **Betty H.C. Cheng**, Professor in Computer Science and Engineering, Michigan State University (http://www.cse.msu.edu/%7Echengb/bio.html)
- **Beijun Shen**, Dept. of Computer Science, East China University of Science and Technology (http://www.iturls.com/~bjshen)
- David S. Rosenblum**,** Professor of Software Systems in the Department of Computer Science at University College London (http://www.cs.ucl.ac.uk/). He is the coordinator of the department's new MSc in Software Systems Engineering and is also Director of London Software Systems, a joint initiative of the Software Systems Group at UCL and the Distributed Software Engineering Group at Imperial College London, where he is also an Honorary Professorial Research Fellow.
- **Jeff Magee**, Professor in Computing, Distributed Software Engineering Section, Department of Computing, Imperial College (http://www-dse.doc.ic.ac.uk/)

8.3.2.1 Invited speakers (with titles of their presentations)

The Future of Software Processes
Barry Boehm
Director and Professor of Center for Software Engineering, Computer Science Department, University of Southern California, U.S.

Software: A Paradigm for the Future
Watts S. Humphrey
Fellow of SEI, Carnegie Mellon University, U.S.

Integrated Software Process & Product Lines
H. Dieter Rombach
Professor of the Department of Computer Science at the University of Kaiserslautern, Germany

Unifying Microprocess and Macroprocess Research

Leon J. Osterweil

Dean and Professor of the College of Natural Sciences & Mathematics, University of Massachusetts Amherst, U.S.

Achieving Software Development Performance Improvement through Process Change

Ross Jeffery

Professor of the School of Computer Science and Engineering at the University of New South Wales, Australia

What Beyond CMMI is Needed to Help Assure Program and Project Success?
Arthur Pyster
Senior Vice President and Director of Systems Engineering and Integration, Federal Segment, SAIC, U.S.

Expanding the Horizons of Software Development Processes: A 3-D Integrated Methodology
Mingshu Li
Director and Professor of the Institute of Software at the Chinese Academy of Sciences, China

Evolving Defect 'Folklore': A Cross-Study Analysis of Software Defect Behavior
Victor R. Basili
Professor of Computer Science at the University of Maryland

Rigorous Software Process for Development of Embedded Systems
Wilhelm Schäfer
Chair and Professor of the International Graduate School of Computer Science and Engineering at the University of Paderborn, Germany

Software are Processes Too
Jacky Estublier
Professor of the French National Research Center (CNRS) in Grenoble, France

8.3.2.2 Program (with names and titles of presentations)

Process Content

Aspect-Oriented Software Development and Software Process
Stanley M. Sutton Jr.
IBM T.J. Watson Research Center, Hawthorne, NY, U.S.

Process Patterns for COTS-Based Development
Ye Yang
Center for Software Engineering, University of Southern California, U.S.

A Value-Based Process for Achieving Software Dependability
Liguo Huang
Computer Science Department, University of Southern California, U.S.

S-RaP: A Concurrent, Evolutionary Software Prototyping Process
Xiping Song, Arnold Rudorfer, Beatrice Hwong, Gilberto Matos, and Christopher Nelson
Siemens Corporate Research Inc., U.S.

A Development Process for Building OSS-Based Applications
Huang Meng, Yang Liguang

Lab for Internet Software Technologies, ISCAS, China
Yang Ye
Center for Software Engineering, University of Southern California, U.S.

A Study on the Distribution and Cost Prediction of Requirements Changes in the Software Life-Cycle
Chengying Mao, Yansheng Lu, and Xi Wang
College of Computer Science and Technology, Huazhong University of Science and Technology, China

A Gradually Proceeded Software Architecture Design Process
Licong Tian, Li Zhang, Bosheng Zhou, and Guanqun Qian
Software Engineering Institute, University of Aeronautics and Astronautics, China

Requirements Engineering Processes Improvement: A Systematic View
Anliang Ning, Hong Hou, Qingyi Hua, Bin Yu and Kegang Hao
Institute of Software Engineering, Northwest University, China

Process Tools and Metrics

Project Management System Based on Work-Breakdown-Structure Process Model
Akira Harada, Satoshi Awane, Yuji Inoya, Osamu Ohno
Hitachi Ltd., Japan
Makoto Matsushita, Shinji Kusumoto, Katsuro Inoue
Osaka University, Japan

Evaluation of the Capability of Personal Software Process Based on Data Envelopment Analysis
Liping Ding, Qiusong Yang, Liang Sun, Jie Tong
Laboratory for Internet Software Technologies, ISCAS, China
Yongji Wang
Laboratory for Internet Software Technologies, ISCAS, China
Key Laboratory for Computer Science, The Chinese Academy of Sciences, China

Spiral Pro: A Project Plan Generation Framework and Support Tool
Jizhe Wang
Laboratory for Internet Software Technologies, ISCAS, China
Graduate School of the Chinese Academy of Sciences, China
Steven Meyers
Center for Software Engineering, University of Southern California, U.S.
Software Process Group, U.S.

Software Testing Process Automation Based on UTP —- A Case Study
Wei Chen, Qun Ying, Yunzhi Xue, and Chen Zhao
Laboratory for Internet Software Technologies, ISCAS, China

Process Management

Software Process Management: Practices in China
Qing Wang and Mingshu Li
Institute of Software, Chinese Academy of Sciences, China

A Framework for Coping with Process Evolution
Brian A. Nejmeh
INSTEP Inc., U.S.
Messiah College, Grantham, Pennsylvania U.S.
William E. Riddle
Solution Deployment Affiliates, U.S.
Fraunhofer IESE, Kaiserslautern, Germany

A Process Improvement Framework and A Supporting Software Oriented to Chinese Small Organizations
Bo Gong
BeiHang University, China
Institute of Command and Technology of Equipment, China
Xingui He
Peking University, China
Weihong Liu
Institute of Command and Technology of Equipment, China

Incremental Workflow Mining based on Document Versioning Information
Ekkart Kindler, Vladimir Rubin, Wilhelm Schäfer
Software Engineering Group, University of Paderborn, Germany

Process Representation and Analysis

Process Technology to Facilitate the Conduct of Science
Leon J. Osterweil, Alexander Wise, Lori Clarke
Department of Computer Science, University of Massachusetts, U.S.
Aaron M. Ellison, Julian L. Hadley, Emery Boose, David R. Foster
Harvard University, U.S.

M(in)BASE: An Upward-Tailorable Process Wrapper Framework for Identifying and Avoiding Model Clashes
David Klappholz
Stevens Institute of Technology, U.S.
Daniel Port
University of Hawaii, U.S.

Process Definition Language Support for Rapid Simulation Prototyping
Mohammad S. Raunak and Leon J. Osterweil
University of Massachusetts, U.S.

Integrated Modeling of Business Value and Software Processes
Raymond Madachy
Center for Software Engineering, Department of Computer Science, University of Southern California, U.S.
Cost Xpert Group, CA

Process Elements: Components of Software Process Architectures
Jesal Bhuta, Barry Boehm
Center for Software Engineering, Computer Science Department, University of Southern California, U.S.
Steve Meyers
Software Process Group, U.S.

Translation of Nets within Nets in Cross-organizational Software Process Modeling
Jidong Ge, Haiyang Hu, Ping Lu, Hao Hu, and Jian Lv
State Key Laboratory for Novel Software Technology, Nanjing University, China

Process Programming to Support Medical Safety: A Case Study on Blood Transfusion
Lori A. Clarke, Yao Chen, George S. Avrunin, Bin Chen, Rachel Cobleigh, Kim Frederick, Elizabeth A. Henneman, and Leon J. Osterweil
University of Massachusetts, U.S.

Experience Report

Experience in Discovering, Modeling, and Reenacting Open Source Software Development Processes
Chris Jensen and Walt Scacchi
Institute for Software Research, University of California, Irvine, U.S.

Application of the V-Modell XT - Report from A Pilot Project
Marco Kuhrmann
Technische Universität München, Germany
Dirk Niebuhr, and Andreas Rausch
Technische Universität Kaiserslautern, Germany

Evolving an Experience Base for Software Process Research
Zhihao Chen
Center for Software Engineering, University of Southern California, U.S.
Daniel Port
University of Hawaii, U.S.

Yue Chen, Barry Boehm
Center for Software Engineering, University of Southern California, U.S.

Status of SPI Activities in Japanese Software - A view from JASPIC
Kouichi Sugahara
Fuji Film Software Corp., Japan
Hideto Ogasawara
TOSHIBA Corp., Japan
Teruyuki Aoyama
Fuji Xerox Corp., Japan
Tetsuya Higashi
TOSHIBA Medical Systems Corp., Japan

Automatically Analyzing Software Processes: Experience Report
Rodion M. Podorozhny
Texas State University, U.S.
Dewayne E. Perry
University of Texas, Austin, U.S.
Leon J. Osterweil
University of Massachusetts, U.S.

A Road Map for Implementing eXtreme Programming
Kim Man Lui and Keith C.C. Chan
Hong Kong Polytechnic University, HK

A Survey of CMM/CMMI Implementation in China
Zhanchun Wu
Laboratory for Internet Software Technologies, Institute of Software, The Chinese Academy of Sciences
David Christensen
One Market Ltd. Co, New Zealand
Mingshu Li
Institute of Software, The Chinese Academy of Sciences
Key Laboratory for Computer Science, The Chinese Academy of Sciences
Qing Wang
Laboratory for Internet Software Technologies, Institute of Software, The Chinese Academy of Sciences

Closing Panel on Directions in Software Process Research: Where Are We Now? Where Should We Go Next?
Chair: Leon J. Osterweil
University of Massachusetts, U.S.

8.3.3 Agile Software Development

The Agile Software Movement attempts to address problems in software purported to involve overly fat processes, too much paperwork, rigid adherence to plans, overly rigorous discipline, bureaucratic burdens, and others. Responding to new environments, some software engineers have posited methods to develop software quicker, cheaper, and better. These software engineers and the methods they promulgate include Kent Beck and eXtreme Programming; Martin Fowler and Refactoring; Ken Schwaber and Scrum; and Jim Highsmith and Adaptive Software Development. More on these methods follows.

eXtreme Programming, developed by Kent Beck et al., was introduced at a conference in June 1999 in Nancy, France. It evokes the most interest of any of the Agile methods. Its flagship is the C3 project at Chrysler, which is mostly an engineering process. It embodies four values: communication, simplicity, feedback, and courage. It promulgates 12 practices: The Planning Game, Small Releases, Metaphor, Simple Design, Testing, Refactoring, Pair Programming, Collective Ownership, Continuous Integration, 40-Hour Week, On-site Customer, and Coding Standards.

Scrum was first used to describe development processes in Japan in 1987. It was first tested in Individual Inc. in 1996 and puts the emphasis on management and control. It purportedly is adaptive, quick, self-organizing, and has few rests—characteristics also shared by eXtreme Programming.

Adaptive Software Development, originally called RADical Software Development, was developed from a mainframe project in 1992 by Jim Highsmith. It was renamed Adaptive Software Development in 1997 in a book published in 2000.

Other methods and their proponents are: Dynamic Systems Development by Arie van Bennekum et al.; Crystal Methods by Alistair Cockburn; Feature Driven Development by Jeff De Luca and Peter Coad; and Lean Development by Bob Charette.

The ***Agile Alliance,*** http://www.agilealliance.com/ , was formed by 17 people who gathered at Snowbird, Utah, in February 2001. They agreed to use the term "Agile" and called themselves "Agilists." They issued the ***Agile Manifesto***, available at http://agilemanifesto.org/. The Manifesto reads: "We are uncovering better ways of developing software by doing it and helping others do it. … Through this work we have come to value:

- individuals and interactions over processes and tools
- working software over comprehensive documentation
- customer collaboration over contract negotiation
- responding to change over following a plan

The ***Manifesto*** encompasses 12 principles:

- satisfy the customer through delivery of valuable software
- welcome changing requirements
- deliver working software frequently
- business people and developers work together daily
- build projects around motivated individuals
- face-to-face conversation
- working software is the primary measure of progress
- promote sustainable development
- continuous attention to technical excellence
- simplicity
- architectures, requirements, and designs emerge
- tune and adjust team behavior regularly

Agile Strategies are the following:

- On project management:
  - use thin, barely sufficient process
  - short, iterative cycles
  - plan is a guide to the future, *not* the future
  - adaptive, rather than predictive planning
  - working software is the primary measure of progress
  - informally documented requirements
  - rely on the collective ability of autonomous teams as the basic problem-solving mechanism
  - real-time communication between team members

- On documentation:
  - remember subsequent maintenance cost
  - keep it lightweight
  - self-explanatory code instead of comments
  - fundamental issue is communication, not documentation
  - primary goal is to develop software
  - create documentation only when you need it
  - update documentation only when it hurts

- On requirements:
  - active stakeholder participation

- expect to gather requirements throughout the entire project
- use the terminology of your users
- use the simplest tools
- keep it fun

- On planning
  - plan from a chaordic perspective
  - view plan as a hypothesis rather than a prediction
  - constant involvement by stakeholders
  - expectation of change
  - rough long-term plans
  - detailed short-term plans
  - use simple tools
  - agile ≠ no plan

- On architecture:
  - don't place architects on pedestals
  - architects should prepare to get their hands dirty with development
  - solve tomorrow's problem tomorrow
  - evolve architecture incrementally, iteratively, allowing it to emerge over time
  - just enough documentation to communicate
  - architectures are proved through concrete experiments

- On design:
  - reject significant effort in up-front design
  - in favor of evolutionary approach
  - the design changes as the program evolves
  - get feedback from coding
  - design emerges

- On coding and testing:
  - pair programming instead of formal inspection or peer review
  - testers should frequently interact with programmers and customers
  - short, informal test plans
  - early testing, even before coding
  - continuous testing, frequent regression
  - automatic testing
  - diverse testers, programmers as testers

- Common themes are:
  - practicing software development in a minimalist manner
  - focusing on generating small, frequent releases

- using mostly collaborative techniques
- emphasizing communication
- advocating simplicity

8.3.4 International Conferences in Software Engineering

As stated on the Web sites for these conferences (http://www.cs.wustl.edu/icse05/Home/index.shtml and http://www.icse-conferences.org/) ICSE is the premier software engineering conference, providing a forum for researchers, practitioners, and educators to present and discuss the most recent innovations, trends, experiences, and concerns in the field of software engineering.

The proceedings of ICSE are available in the ACM Digital Library (http://portal.acm.org/portal.cfm). Through the following link one can find information about the ICSE conference series as well as information about the ICSE Steering Committee: http://www.icse-conferences.org/sc/index.html. The steering committee includes a long list of prominent software engineers, both in academe and industry, worldwide.

The following links provide access to the history of the ICSE Conference, bibliography of ICSE papers, and the list of ICSE "Most Influential Papers":

- History of the ICSE Conference (http://www.icse-conferences.org/sc/history.html)
- Bibliography of ICSE Papers (http://www.icse-conferences.org/sc/bib.html)
- ICSE "Most Influential Papers" (http://www.sigsoft.org/awards/mostInfPapAwd.htm)

8.3.5 Recipients and Title of Most Influential Paper

1989 Marc J. Rochkind: *The Source Code Control System*, NCSE-1, 1975.

1990 William A. Wulf, Ralph L. London, Mary Shaw: An Introduction to the Construction and Verification of Alphard Programs, ICSE-2, 1976.

1991 David Parnas: Designing Software for Ease of Extension and Contraction, ICSE-3, 1978.

1992 Walter Tichy: Software Development Control Based on Module Interconnection, ICSE-4, 1979.

1993 Mark Weiser: Program Slicing, ICSE-5, 1981.

1994 Sol Greenspan, John Mylopoulos, Alex Borgida: Capturing More World Knowledge in the Requirements Specification, ICSE-6, 1982.

1995 David L. Parnas, Paul C. Clements, David M. Weiss: The Modular Structure of Complex Systems, ICSE-7, 1984.

1996 Sam Redwine Jr., William Riddle: Software Technology Maturation, ICSE-8, 1985.

1997 Lee Osterweil: Software Processes are Software Too, ICSE-9, 1987.

Manny Lehman: Process Models, Process Programs, Programming Support, ICSE-9, 1987.

1998 David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtul-Trauring: Statemate: A Working Environment for the Development of Complex Reactive Systems, ICSE-10, 1988.

1999 Dewayne Perry: The Inscape Environment, ICSE-11, 1989.

2000 No award.

2001 Robert Balzer: Tolerating Inconsistency, ICSE-13, 1991.

2002 David S. Rosenblum: Towards a Method of Programming with Assertions, ICSE-14, 1992.

2003 Bashar Nuseibeh, Jeff Kramer, Anthony Finkelstein: Expressing the Relationships between Multiple Views in Requirements Specification, ICSE-15, 1993.

2004 Robert Allen, David Garlan: Formalizing Architectural Connection, ICSE-16, 1994.

2005 Michael Jackson, Pamela Zave: Deriving Specifications from Requirements: An Example, ICSE-17, 1995.

An analysis and commentary on the impact of these influential papers will be the subject of a future report.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	December 2005	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Results of SEI Independent Research and Development Projects and Report on Emerging Technologies and Technology Trends	F19628-00-C-0003

6. AUTHOR(S)

Sagar Chaki; Rosann W. Collins; Peter Feiler; John Goodenough; Aaron Greenhouse; Jorgen Hansson; Alan R. Hevner; John Hudak; Angel Jordan; Rick Kazman; Richard C. Linger; Mark G. Pleszkoch; Stacy J. Prowell; Natasha Sharygina; Kurt C. Wallnau; Gwen Walton; Chuck Weinstock; & Lutz Wrage

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

CMU/SEI-2005-TR-020

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

ESC-TR-2005-020

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	117

13. ABSTRACT (MAXIMUM 200 WORDS)

Each year, the Software Engineering Institute (SEI) undertakes several Independent Research and Development (IR&D) projects. These projects serve to (1) support feasibility studies investigating whether further work by the SEI would be of potential benefit, and (2) support further exploratory work to determine whether there is sufficient value in eventually funding the feasibility study work as an SEI initiative. Projects are chosen based on their potential to mature and/or transition software engineering practices, develop information that will help in deciding whether further work is worth funding, and set new directions for SEI work. This report describes the IR&D projects that were conducted during fiscal year 2005 (October 2004 through September 2005). In addition, this report provides information on what the SEI has learned in its role as a technology scout for developments over the past year in the field of software engineering.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software engineering research and development	117

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL